AD-A102 308 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
IMPLEMENTATION OF PROCESS MANAGEMENT FOR A SECURE ARCHIVAL STOR--ETC(U)
MAR 81 A R STRICKLER

UNCLASSIFIED NL

1 of 3

AD
A102308

LEVEL

2

AD A102308

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

IMPLEMENTATION OF PROCESS MANAGEMENT
FOR A SECURE ARCHIVAL STORAGE SYSTEM

by

Anthony Ross Strickler

March 1981

Thesis Advisor: R. R. Schell

81 8 03 024

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A102308	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Implementation of process management for a secure archival storage system. (9)		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis, March 1981
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Anthony Ross/Strickler		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgreduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940 (11)		12. REPORT DATE Mar[illegible]81
		13. NUMBER OF PAGES 227
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 228		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer security, Process Management, Distributed operating systems, Traffic Controller process scheduling, event counts and sequencers

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis presents an implementation of process management for a security kernel based secure archival storage system (SASS). The implementation is based on a family of secure, distributed, multi-microprocessor operating systems designed to provide multilevel internal security and controlled sharing of data among authorized users. Process scheduling is effected by one half of a two level Traffic Controller that binds processes to virtualized processors. Inter-process communication mechanisms for

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

Unclassified 1
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

25145(

synchronization, mutual exclusion, and message passing among processes are provided by utilization of eventcount and sequencer primitives. The implementation structure is based upon levels of abstraction and is loop free to permit future expansion to more complex members to the design family. Implementation was completed on the ADVANCED MICRO COMPUTERS Am 96/4116 AmZ8002 16 Bit MonoBoard Computer.

Implementation of Process Management for a Secure Archival Storage System

by

Anthony R. Strickler
Captain, United States Army
B.S., United States Military Academy, 1973

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1981

Author [illegible]

Approved by: Roger R. Schell
Thesis Advisor

Lyle A. Cox Jr.
Second Reader

[illegible]
Chairman, Department of Computer Science

[illegible]
Dean of Information and Policy Sciences

ABSTRACT

This thesis presents an implementation of process management for a security kernel based secure archival storage system (SASS). The implemention is based on a family of secure, distributed, multi-microprocessor operating systems designed to provide multilevel internal security and controlled sharing of data among authorized users. Process scheduling is effected by one half of a two level Traffic Controller that binds processes to virtualized processors. Inter-process communication mechanisms for syncronization, mutual exclusion, and message passing among processes are provided by utilization of eventcount and sequencer primitives. The implementation structure is based upon levels of abstraction and is loop free to permit future expansion to more complex members of the design family. Implementation was completed on the ADVANCED MICRO COMPUTERS Am 96/4116 AmZ8002 16 bit Monoboard Computer.

TABLE OF CONTENTS

I. INTRODUCTION....................................10

A. BACKGROUND....................................11

B. BASIC CONCEPTS/DEFINITIONS....................13

1. Process....................................13

2. Information Security.......................15

3. Segmentation...............................20

4. Protection Domains.........................22

5. Abstraction................................23

C. THESIS STRUCTURE..............................24

II. SECURE ARCHIVAL STORAGE SYSTEM DESIGN.............26

A. BASIC SASS OVERVIEW...........................26

B. SUPERVISOR....................................29

1. File Manager Process.......................31

2. Input/Output Process.......................32

C. GATE KEEPER...................................33

D. DISTRIBUTED KERNEL............................35

1. Segment Manager............................36

2. Event Manager..............................38

3. Non-Discretionary Security Module..........39

4. Traffic Controller.........................39

5. Inner Traffic Controller...................43

6. Distributed Memory Manager.................47

E. NON-DISTRIBUTED KERNEL....................................48

1. Memory Manager Process....................................49

F. SYSTEM HARDWARE...52

G. SUMMARY...55

III. IMPLEMENTATION ISSUES...................................56

A. DATABASE INITIALIZATION...................................56

1. Inner Traffic Controller Initialization...................57

2. Traffic Controller Initialization.........................62

3. Additional Initialization Requirements....................63

B. PREEMPT INTERRUPTS..64

1. Physical Preempt Handler..................................64

2. Virtual Preempt Handler...................................66

C. IDLE PROCESSES..71

D. ADDITIONAL KERNEL REFINEMENTS.............................72

E. SUMMARY...74

IV. PROCESS MANAGEMENT IMPLEMENTATION.........................76

A. EVENT MANAGER MODULE......................................78

1. Support Procedures..79

2. Read..81

3. Ticket..81

4. Await...82

5. Advance...82

B. TRAFFIC CONTROLLER MODULE.................................83

1. TC_GETWORK..84

2. TC_AWAIT..85

3. TC_ADVANCE..86

4. Virtual_Preempt_Handler....................91

5. Remaining Procedures.......................91

C. DISTRIBUTED MEMORY MANAGER MODULE..............92

1. MM_Read_Eventcount.........................92

2. MM_Advance.................................93

3. MM_Ticket..................................94

4. MM_Allocate................................94

D. GATE KEEPER MODULES............................96

1. User_Gate Module...........................97

2. Kernel_Gate_Keeper Module..................98

E. SUMMARY..101

V. CONCLUSION...102

A. FOLLOW ON WORK.................................103

APPENDIX A--EVENT MANAGER LISTINGS....................104

APPENDIX B--TRAFFIC CONTROLLER LISTINGS...............115

APPENDIX C--DISTRIBUTED MEMORY MANAGER LISTINGS.......146

APPENDIX D--GATE KEEPER LISTINGS......................168

APPENDIX E--BOOTSTRAP LOADER LISTINGS.................177

APPENDIX F--LIBRARY FUNCTION LISTINGS.................190

APPENDIX G--INNER TRAFFIC CONTROLLER LISTINGS.........195

LIST OF REFERENCES....................................224

INITIAL DISTRIBUTION LIST.............................226

LIST OF FIGURES

1. SASS System..27
2. System Overview (Dual Host)........................30
3. Known Segment Table (KST)..........................37
4. Active Process Table (APT).........................41
5. Virtual Processor Table (VPT)......................45
6. Extended Instruction Set...........................51
7. Kernel Databases...................................52
8. Memory Management Unit (MMU) Image.................54
9. Initial Process Stack..............................68
10. Implementation Module Structure...................77
11. Advance Algorithm.................................87
12. Program Status Area...............................99

ACKNOWLEDGEMENT

This research is sponsored in part by the Office of Naval Research Project Number NR 337-0x5, monitored by Mr. Joel Trimble.

I am indebted to a number of people for the valuable support that I have received in this thesis effort. My thesis advisor, Lt. Col. Roger Schell, provided a wealth of knowledge and many hours of patient counseling. This thesis could not have been written without his enthusiastic guidance.

Thanks are also extended to my reader, Professor Lyle Cox, for his assistance and concern. Tim Wells sacrificed precious time in the final days of his thesis work to introduce me to the Zilog Developemental System providing the programming environment for this implementation. Gary Baker, Bob McDonnell, and Mike Williams provided excellent technical assistance, especially in helping me with the many hardware problems that I encountered in working with a new and unfamiliar system.

Finally, special thanks and appreciation go to my wife, Brenda, and my children, Christopher and Mark for their undying love, patience, and understanding. They always support me whatever the endeavor.

I. INTRODUCTION

This thesis addresses the implementation of process management functions for the Secure Archival Storage System or SASS. This system is designed to provide multilevel secure access to information stored for a network of possibly dissimilar host computer systems and the controlled sharing of data amongst authorized users of the SASS. Effective process management is essential to insure efficient use and control of the system.

Among the major accomplishments of the work reported here are the inclusion of provisions for efficient process creation and management. These functions are provided through the establishment of a system Traffic Controller and the creation of a virtual interrupt structure. An effective mechanism for inter-process communication and synchronization is realized through an Event Manager that makes use of uniquely identified segments supported by eventcount and sequencer primitives. A hardware controlled two domain operational environment is created with the necessary interfacing between domains provided by a software "gate" mechanism. Additional support is provided through considerable work in the area of database initialization and a technique for limited dynamic memory allocation.

This implementation was completed on the commercial AMC Am96/4116 MonoBoard Computer with a standard Multibus interface.

A. BACKGROUND

The brief history of digital computers has been characterized by rapid advances in hardware technology and a continual increase in the number and variety of its applications. The advent of the microprocessor has enabled virtually every level of our society to make use of computer resources. Today's "desk top" microcomputers, costing less than a thousand dollars, have more computing power than the "giant" computers of the early 1950's that cost hundreds of times that amount.

These rapid advances in computer hardware technology have reversed the economics of the computer design environment. While hardware costs have decreased, the relative costs of the software required to effectively utilize this hardware has steadily increased until it now dominates the overall cost of a computer system. This economic reversal requires that developed software be logical, easy to read, relatively maintenance free, and easy to debug. Unfortunately, microcomputer operating systems and applications software tend to be highly specialized, thus failing to reasonably exploit the potential of the microprocessor.

As the usage of computers has expanded, expecially in the area of sensitive information handling, the need for information security has received greater recognition. While ad-hoc attempts have been made to provide internal computer security on larger systems, the problem of information security on microprocessors has been largely ignored to date.

In an attempt to address the above problems, O'Connell and Richardson [1] outlined a high level design for a microprocessor based secure operating system. The goal of this design was to provide information security, distributed processing, multiple protection domains, configuration independence, multiprocessing, and multiprogramming. Since all computer applications do not require such a broad and general operating system, the design provided for a family of operating systems. This allows a member of the family to incorporate only the subset of family functions needed for its specific application, while providing for future expansion. The SASS is a member of this operating system family.

A brief history of prior work done on the SASS is now provided. Parks [2] provided the design for the SASS Supervisor. The actual implementation of the Supervisor design has not been addressed to date. The initial design of the SASS Security Kernel was completed by Coleman [3]. The works of O'Connell and Richardson [1], Parks [2], and

Coleman [3] are available as a single publication from NTIS and DDC in a report prepared by Schell and Cox [21]. Further refinements of the Kernel design and partial Kernel implementation has been accomplished in three additional thesis efforts. Moore and Gary [4] provided the detailed design and partial implementation of the Memory Manager module. Design refinements for the Inner Traffic Controller and Traffic Controller modules as well as implementation of the Inner Traffic Controller was provided by Reitz [5]. Wells [6] provided implementation of the Segment Manager and Non-Discretionary Security modules as well as partial implementation of distributed Memory Manager functions. These design and implementation efforts provided the basis for the work described here.

B. BASIC CONCEPTS/DEFINITIONS

This section provides an overview of several concepts essential to the SASS design. Readers familiar with SASS or with secure operating system principles may wish to skip to the next section.

1. Process

The notion of a process has been viewed in many ways in computer science literature. Organick [7] defines a process as a set of related procedures and data undergoing execution and manipulation, respectively, by one of possibly several processors of a computer. Madnick and Donovan [8] view a process as the locus of points of a processor executing a collection of programs. Reed [9] describes a process as the sequence of actions taken by some processor. In other words, it is the past, present, and future "history" of the states of the processor. In the SASS design, a process is viewed as a logical entity entirely characterized by an address space and an execution point. A process' address space consists of the set of all memory locations accessible by the process during its execution. This may be viewed as a set of procedures and data related to the process. The execution point is defined by the state of the processor at any given instant of process execution.

As a logical entity, a process may have logical attributes associated with it, such as a security access class, a unique identifier, and an execution state. This notion of logical attributes should not be confused with the more typical notion of physical attributes, such as location in memory, page size, etc. In SASS, a process is given a security access class, at the time of its creation, to specify what authorization it possesses in terms of information access (to be discussed in the next section). It is also given a unique identifier that provides for its identification by the system and is utilized for interaction among processes. A process may exist in one of three execution states: 1) running, 2) ready, and 3) blocked. In order to execute, a process must be mapped onto (bound to) a physical processor in the system. Such a process is said to be in the "running" state. A process that is not mapped onto a physical processor, but is otherwise ready to execute, is in the "ready" state. A process in the "blocked" state is waiting for some event to occur in the system and cannot continue execution until the event occurs. At that time, the process is placed into the ready state.

2. Information Security

There is an ever increasing demand for computer systems that can provide controlled access to the data it stores. In this thesis, "information security" is defined as the process of controlling access to information based upon proper authorization. The critical need for information security should be clear. Banks and other commercial enterprises risk the theft or loss of funds. Insurance and credit companies are bound by law to protect the private or otherwise personal information they maintain on their customers. Universities and scientific institutions must prevent the unauthorized use of their often over-burdened systems. The Department of Defense and other government agencies must face the very real possibility that classified information is being compromised or that weapon systems are being tampered with. In fact, security related problems can be found at virtually every level of computer usage.

In the past, attempts have been made to identify the security weakness of computer systems by trial and error and then fix them. However, Schell [10] has shown that security cannot be "added on" to an existing system with any degree of confidence that the resulting security system is impregnable. Security must be explicitly designed into a system from first principles. The key to achieving provable information security is realized in the concept of the "security kernel." Schell [11] provides a detailed discussion of the use of this concept in the methodical design of system security.

The security of computer systems processing sensitive information can be achieved by two means: external security controls and internal security controls. In the first case, security is achieved by encapsulating the computer and all its trusted users within a single security perimeter established by physical means (e.g., armed guards, fences, etc.) This means of security is often undesirable due to its added cost of implementation, the inherent risk of error-prone manual procedures, and the problem of trustworthy but error-prone users. Also, since all security controls are external to the computer system, the computer is incapable of securely handling data at differing security levels or users with differing degrees of authorization. This restriction greatly limits the utility of modern computers. Internal security controls rely upon the computer system to internally distinguish between multiple levels of information classification and user authorization. This is clearly a more desirable and flexible approach to information security. This does not mean, however, that external security is not needed. The optimal approach would be to utilize internal security controls to maintain information security and external security controls to provide physical protection of our system against sabotage, theft, or destruction. The primary concern of this thesis is information security and will therefore center its discussion on the achievement of information security through implementation of the security kernel concept.

One might argue that a "totally secure" computer system is one that allows no access to its classified or otherwise sensitive information. Such a system would not be of much value to its users. Therefore, when we say that a system provides information security, it is only secure with respect to some specific external security policy established by laws, directives, or regulations. There are two distinct aspects of security policy: non-discretionary and discretionary. Each user (subject) of the system is given a label denoting what classification or level of access the user is authorized. Likewise, all information or segments (objects) within the system are labelled with their classification or level of sensitivity. The non-discretionary security mechanism is responsible for comparing the authorization of a subject with the classification of an object and determining what access, if any, should be granted. The DOD security classification system provides an example of the non-discretionary security policy and is the policy implemented in SASS. The discretionary security policy is a refinement of the non-discretionary policy. As such, it adds a higher degree of restriction by allowing a subject to specify or restrict who may have access to his files. It must be emphasized that the discretionary policy is contained within the non-discretionary policy and in no way undermines or substitutes for it. This prevents a subject from granting access that would violate the non-discretionary policy. An example of discretionary security is provided by the DOD "need to know" policy. In SASS, the discretionary policy is implemented within the supervisor [2] by means of an Access Control List (ACL). There is an ACL maintained for every file in the system, which provides a list of all users authorized access to that file. Every attempt by a user to access a file is first checked against the ACL and then checked against the non-discretionary security policy. The "least" or "most restrictive" access found in these checks is then granted to the user.

The relationship between the labels associated with the subject's access class (sac) and the object's access class (oac) is defined by a lattice model of secure information flow [12] as follows ("¦" denotes "no relationship"):

1. sac = oac, read and write access permitted
2. sac > oac, read access permitted
3. sac < oac, write access permitted
4. sac ¦ oac, no access permitted

In order to understand how these access levels are determined, it is necessary to gain an awareness of and consideration for several basic security properties.

The "Simple Security Property" deals with "read" access. It states that a subject may have read access only to those object's whose classification is less than or equal to the classification of the subject. This prevents a subject from reading any object possessing a classification higher than his own.

The "Confinement Property" (also known as "*-property") governs "write" access. It states that a user may be granted write access only to those objects whose classification is greater than or equal to the classification of the subject. This prevents a user from writing information of a higher classification (e.g., Secret) into a file of a lower classification (e.g., Unclassified). It is noted that while this property allows a user to write into a file possessing a classification higher than his own, it does not allow him access to any of the data in that file. The SASS design does not allow a user to "write up" to higher classified files. Therefore, in SASS, "sac < oac" denotes "no access permitted."

The "Compatibility Property" deals with the creation of objects in a hierarchical structure. In SASS, objects (segments) are hierarchically organized in a tree structure. This structure consists of nodes with a root node from which the tree eminates. The Compatibility Property states that the classification of objects must be non-decreasing as we move down the hierarchical structure. This prevents a parent node from creating a child node of a lower classification.

Several prerequisites must be met in order to insure that the security kernel design provides a secure environment. Firstly, every attempt to access data must invoke the Kernel. In addition, the Kernel must be isolated and tamperproof. Finally, the Kernel design must be verifiable. This implies that the mathematical model, upon which the Kernel is based, must be proved secure and that the Kernel is shown is to correctly implement this model.

3. Segmentation

Segmentation is a key element of a security Kernel based system. A segment can be defined as a logical grouping of information, such as a procedure, file or data area [8]. Therefore, we can redefine a process′ address space as the collection of all segments addressable by that process. Segmentation is the technique applied to effect management of those segments within an address space. In a segmented environment, all references within an address space require two components: 1) a segment specifier (number) and 2) the location (offset) within the segment.

A segment may have several logical and physical attributes associated with it. The logical attributes may include the segment's classification, size, or permissable access (read, write, or execute). These logical attributes allow a segment to nicely fit the definition of an object within the security kernel concept, and thus provide a means for the enforcement of information security. A segment's physical attributes include the current location of the segment, whether or not the segment resides in main memory or secondary storage, and where the segment's attributes are maintained by a segment descriptor. The segment descriptors for each segment in a process' address space are contained within a Descriptor Segment (viz., the MMU Image in SASS) to facilitate the memory management of that address space.

Segmentation supports information sharing by allowing a single segment to exist in the address spaces of multiple processes. This allows us to forego the maintenance of multiple copies of the same segment and eliminates the possibility of conflicting data. Controlled access to a segment is also enforced, since each process can have different attributes (read/write) specified in its segment descriptor. In the implementation of SASS, any segment which is shared, but has "read only" access by every process sharing it, is placed in the processor local memory supporting each of these processes rather than in the global memory. This implies the maintenance of multiple copies of some shared segments. It is noted that the problem of "conflicting data" is avoided since this only applies to read only segments. This apparent waste of memory and nonuse of existing sharing facilities is justified by a design decision to provide maximum reduction of bus contention among processors accessing global memory. This reduction in bus contention is considered to be of more importance than the saving of memory space provided by single copy sharing of read only segments. This decision is also well supported by the occurrence of decreasing memory costs, which we have experienced in terms of high speed bus costs.

4. Protection Domains

The requirement for isolating the Kernel from the remainder of the system is achieved by dividing the address space of each process into a set of hierarchical domains or protection rings [13]. O'Connell and Richardson [1] defined three domains in the family of secure operating systems: the user, the supervisor, and the kernel. Only two domains are actually necessary in the SASS design since it does not provide extended user applications. The Kernel resides in the inner or most privileged domain and has access to all segments in an address space. System wide data bases are also maintained within the Kernel domain to insure their accessibility is only through the Kernel. The Supervisor exists in the outer or least privileged domain where its access to data or segments within an address space is restricted.

While protection domains may be created through either hardware or software mechanisms, a hardware implementation provides much greater efficiency. Current microprocessor technology only provides for the implementation of two domains. This two domain restriction does not support O'Connell and Richardson's complete family design, but it is sufficient to allow hardware implementation of the ring structure required by the SASS subset.

5. Abstraction

Dijkstra [14] has shown that the notion of abstraction can be used to reduce the complexity of a problem by applying a general solution to a number of specific cases. A structure of increasing levels of abstraction provides a powerful tool for the design of complex systems and generally leads to a better design with greater clarity and fewer errors.

Each level of abstraction creates a virtual hierarchical machine [8] which provides a set of "extended instructions" to the system. A virtual machine cannot make calls to another virtual machine at a higher level of abstraction and in fact is unaware of its existence. This implies that a level of abstraction is independent of any higher levels. This independence provides for a loop-free design. Additionally, a higher level may only make use of the resources of a lower level by applying the extended instruction set of the lower level virtual machine.

Therefore, once a level of abstraction is created, any higher level is only interested in the extended instruction set it provides and is not concerned with the details of its implementation. In SASS, once a level of abstraction is created for the physical resources of the system, these resources become "virtualized" making the higher levels of the design independent of the physical configuration of the system.

C. THESIS STRUCTURE

This thesis describes the implementation of the process management functions for the SASS. The design base for this implementation evolved from the secure family of operating systems designed by O'Connell and Richardson [1]. The programming language utilized in this implementation was PLZ/ASM assembly code [20].

Chapter I provided an introduction to the Secure Archival Storage System and a discussion of the basic concepts which underlie a secure operating system environment.

Chapter II will provide a discussion of the SASS design. An overview of the entire SASS system is presented along with more detailed description of the modules comprising SASS and their associated databases.

Chapter III discusses the issues bearing on this implementation and the refinements made to previous SASS related work. A discussion concerning the initialization of the databases utilized by the current SASS demonstration is also presented.

Chapter IV presents the implementation of process management (viz., the Traffic Controller, Event Manager, Distributed Memory Manager, and Gate Keeper stub modules). A description of design and implementation criteria, and decisions made during implementation are also discussed in this chapter.

Chapter V provides the conclusions reached, the status of the research, and recommendations relative to the continuation and extension of this work.

The appendices include the PLZ/ASM code for the modules implemented and refined. The complete program listings for the Secure Archival Storage System may be obtained from a report prepared by Schell and Cox [22].

II. SECURE ARCHIVAL STORAGE SYSTEM DESIGN

This chapter provides an overview of the SASS in its current design state. The intent of this summary is threefold. First, it is intended to provide an overall understanding of the SASS itself. Secondly, it will provide an interrelationship between the work done in this thesis and previous work performed on SASS. Lastly, it provides a current base upon which further SASS development can occur.

A. BASIC SASS OVERVIEW

The purpose of the Secure Archival Storage System is to provide a secure "data warehouse" or information pool which can be accessed and shared by a variable set of host computer systems possessing differing security classifications. The primary goals of the SASS design are to provide information security and controlled sharing of data among system users.

Figure 1 provides an example of a possible SASS usage. The system is used exclusively for managing an archival storage system and does not provide any programming services to its users. Thus the users of the SASS may only create, store, retrieve, or modify files within the SASS. The host computers are hardwired to the system via the I/O ports of the Z8001 with each connection having a fixed security

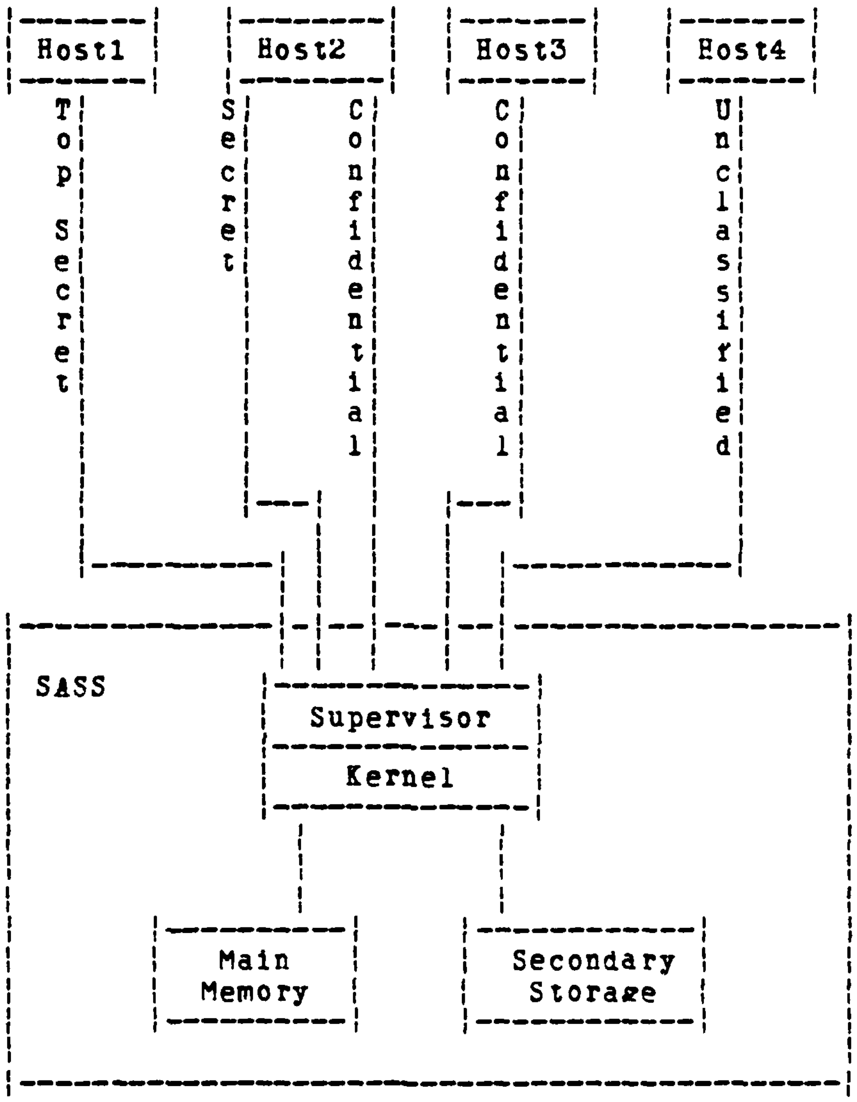

Figure 1. SASS System

classification. Each host must have a separate connection for each security level it wishes to work on (It is important to note that Figure 1 only represents the logical interfacing of the system. Specifically, the actual connection with the host system must be interfaced with the Kernel as the I/O instructions for the port are privileged). In our example, Host #1 can create and modify only Top Secret files, but it can read files which are Top Secret, Secret, Confidential, or Unclassified. Likewise, Host #2 can create or modify secret files, using its secret connection or confidential files, using its confidential connection. Host #2 cannot create or modify Top Secret or Unclassified files.

In order to provide information security and controlled sharing of files, the SASS operates in two domains: (1) the Supervisor domain and (2) the Kernel domain. The SASS achieves this desired environment through a distributed operating system design which consists of two primary modules: the Supervisor and the Security Kernel. Each host system connected to the SASS has associated with it two processes within the SASS which perform the data transfer and file management on behalf of that host. The host computer communicates directly with its own I/O process and File Manager process within the SASS.

We can use our notion of abstraction to present a system overview of the SASS. The SASS consists of four primary levels of abstraction:

Level 3-The Host Computer Systems

Level 2-The Supervisor

Level 1-The Security Kernel

Level Ø-The SASS Hardware

A pictorial representation of this abstract system overview is presented in Figure 2. This representation is limited to a dual host system for clarity and space restrictions. Note that the Gate Keeper module is in actuality the logical boundary between levels one and two and as such will be described separately.

Level 3, the host computer systems, of SASS has already been addressed. It should be noted that the SASS design makes no assumptions about the host computer systems. Therefore each host may be of a different type or size (i.e.- micro, mini, or maxi-computer system). Furthermore, the necessary physical security of the host systems and their respective data links with the SASS is assumed.

B. SUPERVISOR

Level 2 of the SASS system is composed of the Supervisor domain. As already stated, the SASS consists of two domains. The actual implementation of these domains was greatly simplified since the Z8ØØ1 microprocessor provides two modes of execution. The system mode, with which the Kernel was implemented, provides access to all machine instructions and

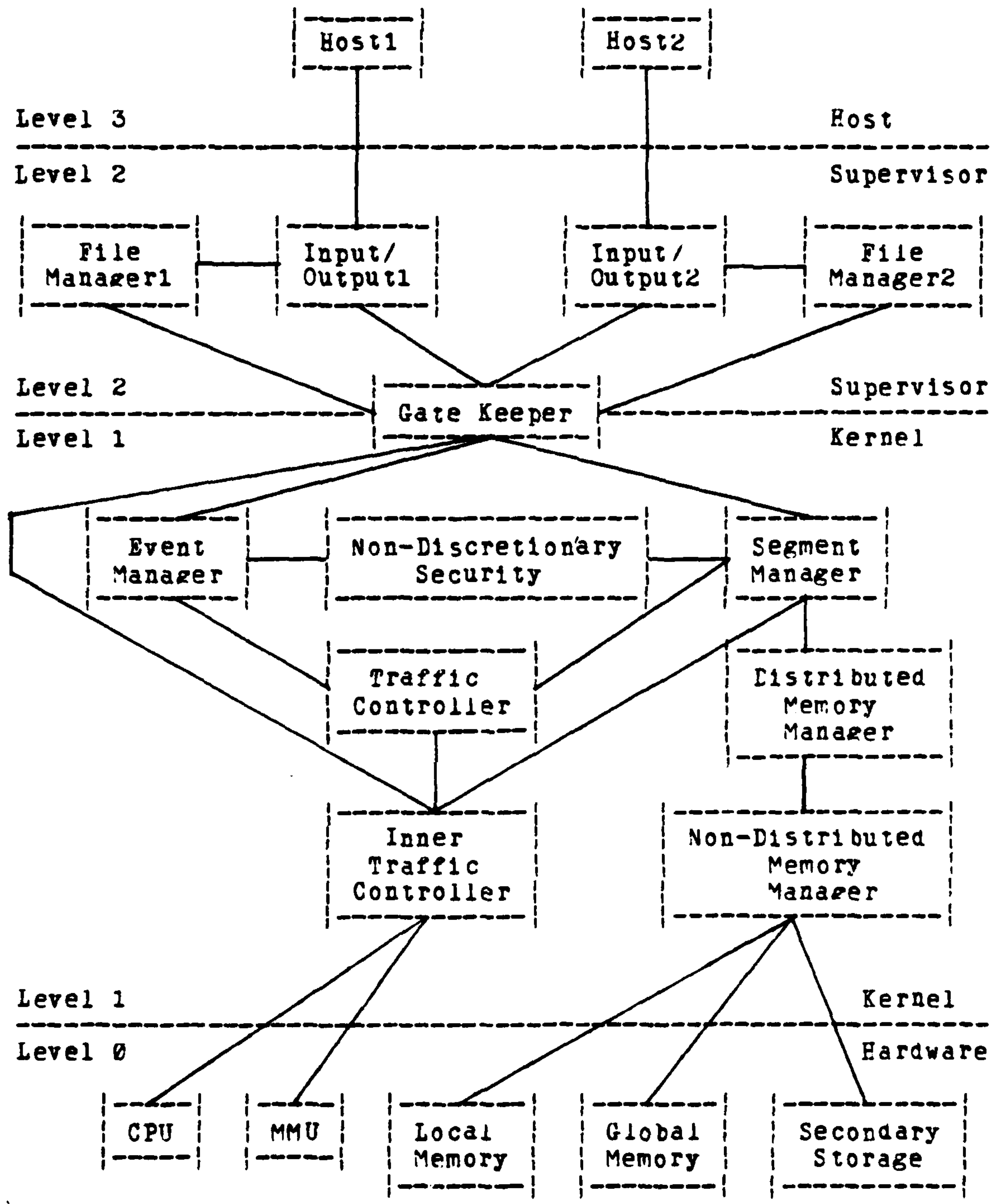

Figure 2. System Overview (Dual Host)

all segments within the system. The normal mode, with which the Supervisor was implemented, only provides access to a limited subset of machine instructions and segments within the system. Therefore, the Supervisor operates in an outer or less privileged domain than the Kernel.

The purpose of the Supervisor is to manage the data link between the host computer systems and the SASS by means of Input/Output control, and to create and manage the file hierarchy of each host within the SASS. These functions are accomplished via an Input/Output (I/O) process and a File Manager (FM) process within the Supervisor. A separate FM and I/O process are created and dedicated to each host at the time of system initialization.

1. File Manager Process

The FM process directs the interaction between the host computer systems and the SASS. It interprets all commands received from the Host computer and performs the necessary action upon them through appropriate calls to the Kernel. The primary functions of the FM process are the management of the Host's virtual file system and the enforcement of the discretionary security policy.

The virtual file system of the Host is viewed as a hierarchy of files which are implemented in a tree structure. The five basic actions which may be initiated upon a file at this level are: 1) to create a file, 2) to delete a file, 3) to read a file, 4) to store a file, and 5) to modify a file. The FM process utilizes a FM Known Segment Table (FM_KST) as the primary database to aid in this management.

The FM process maintains an Access Control List (ACL) through which it enforces the discretionary security in SASS. The FM process initializes an ACL for every file in its Host's file system. The ACL is merely a list of all users that are authorized to access that file. The ACL is checked upon every attempt to access a file to determine its authorization. The user (host computer) directs the FM process as to what entries or deletions should be made in the ACL, and as such, specifies who he wishes to have access to his file. As noted earlier, discretionary security is a refinement to the Non-Discretionary Security Policy and therefore can only be utilized to add further access restrictions to those provided by the Non-Discretionary Security. This prevents a user from granting access to a file to someone who otherwise would not be authorized access.

2. <u>Input/Output Process</u>

The I/O process is responsible for managing the input and output of all data between the host computer systems and the SASS. The I/O process is subservient to the FM process and receives all of its commands from it. Data is transferred between the SASS and Host Computer systems in fixed size "packets". These packets are broken up into three basic types: 1) a synchronization packet, 2) a command packet, and 3) a data packet. In order to insure reliable transmission and receipt of packets between the Host computer and the SASS, there must exist a protocol between them. Parks [2] provides a more detailed description of these packets, and a possible multi-packet protocol.

C. GATE KEEPER

The primary objective of the gate keeper is to isolate the Kernel and make it tamperproof. This goal is accomplished by reason of a software ring crossing mechanism provided by the gate keeper. In terms of SASS, this notion of "ring-crossing" is merely the transition from the Supervisor domain to the Kernel domain. As noted earlier, the gate keeper establishes the logical boundary between the Supervisor and the Kernel, and as a matter of course, it provides a single software entry point (enforced by hardware) into the Kernel. Therefore, any call to the Kernel must first pass through the gate keeper.

The gate keeper acts as a trap handler. Once it is invoked by a user (Supervisor) process, the hardware preempt interrupts are masked, and the user process' registers and stack pointer are saved (within the kernel domain). It then takes the argument list provided by the caller and validates these passed parameters to insure their correctness. To aid in the validation of these parameters, the gate keeper utilizes the Parameter Table as a database. The Parameter table contains all of the permitted functions provided by the Kernel. These relate directly to the extended instruction set (viz., Supervisor calls) provided by the Kernel (these extended instructions will be described in the next section). If an invalid call is encountered by the gate keeper, an error code is returned, and the Kernel is not invoked. If a valid call is encountered by the gate keeper, the arguments and control are passed to the appropriate Kernel module.

Once the Kernel has completed its action on the user request, it passes the necessary parameters and control back to the gate keeper. At this point, the gate keeper determines if any software virtual preempt interrupts have occurred. If they have, then the virtual preempt handler is invoked vice the Kernel being exited (virtual interrupt structure is discussed in chapter III). Correspondingly, if a software virtual preempt has not occurred, then the return arguments are passed to the user process. The user process' registers and stack pointer (viz., its execution point) are restored and control returned to the Supervisor domain. A detailed description of the Gate Keeper interface and implementation is provided in chapter IV.

D. DISTRIBUTED KERNEL

Level 1 of our abstract view of SASS consists of two components: the distributed Kernel and the non-distributed Kernel. These two elements comprise the Security Kernel of the SASS. The Security Kernel has two primary objectives: 1) the management of the system's hardware resources, and 2) the enforcement of the non-discretionary security policy. It executes in the most privileged domain (viz., the system mode of the Z8001) and has access to all machine instructions. The following section will provide a brief description of the distributed Kernel, its components, and the extended instruction set it provides. A discussion of the non-distributed Kernel will be given in the next section.

The distributed Kernel consists of those Kernel modules whose segments are contained (distributed) in the address space of every user (Supervisor) process. Thus, in effect, the distributed Kernel is shared by all user processes in the SASS. The distributed Kernel is composed of the Segment Manager, the Event Manager, the Non-Discretionary Security Module, the Traffic Controller, the Inner Traffic Controller, and the Distributed Memory Manager Module. The Segment Manager and the Event Manager are the only "user visible" modules in the distributed Kernel. In other words, the set of extended instructions available to user processes invoke either the Segment Manager or the Event Manager.

1. Segment Manager

The objective of the Segment Manager is the management of a process' segmented virtual storage. The Segment Manager is invoked by calls from the Supervisor domain via the gate keeper. Calls to the Segment Manager are made by means of six extended instructions provided by the segment manager. These extended instructions (viz., entry points) are: 1) CREATE_SEGMENT, 2) DELETE_SEGMENT, 3) MAKE_KNOWN, 4) TERMINATE, 5) SM_SWAP_IN, and 6) SM_SWAP_OUT. The extended instructions CREATE_SEGMENT and DELETE_SEGMENT add and remove segments from the SASS. MAKE_KNOWN and TERMINATE add and remove segments from the address space of a process. Finally, SM_SWAP_IN and SM_SWAP_OUT move segments from secondary storage to main storage and vice versa.

The primary database utilized by the Segment Manager is the Known Segment Table (KST). A representation of the structure of the KST is provided in figure 3. The KST is a process local database that contains an entry for every segment in the address space of that process. The KST is indexed by segment number with each record of the KST containing descriptive information for a particular segment. The KST provides a mapping mechanism by which the segment number of a particular segment can be converted into a unique handle for use by the Memory Manager. The Memory Manager will be discussed in the next section.

```
|----Segment #
|    |-----------|-------|--------|-------|--------|---------|--------|
|    |MM Handle  | Size  | Acess  | In    | Class  | Mentor  | Entry  |
|    |           |       | Mode   | Core  |        | Seg No  | Number |
|    |-----------|-------|--------|-------|--------|---------|--------|
|    |           |       |        |       |        |         |        |
|    |-----------|-------|--------|-------|--------|---------|--------|
|    |           |       |        |       |        |         |        |
V    |-----------|-------|--------|-------|--------|---------|--------|
     |           |       |        |       |        |         |        |
     |-----------|-------|--------|-------|--------|---------|--------|
     |           |       |        |       |        |         |        |
     |-----------|-------|--------|-------|--------|---------|--------|
     |           |       |        |       |        |         |        |
     |-----------|-------|--------|-------|--------|---------|--------|
     |           |       |        |       |        |         |        |
     |-----------|-------|--------|-------|--------|---------|--------|
```

Figure 3. Known Segment Table (KST)

2. Event Manager

The purpose of the Event Manager is the management of event data which is associated with interprocess communications within the SASS. This event data is implemented by means of eventcounts (a synchronization primitive discussed by Reed [15]). The Event Manager is invoked, via the Gate Keeper, by user processes residing in the Supervisor domain. There are two eventcounts associated with every segment existing in the Supervisor domain. These eventcounts (viz., Instance 1 and Instance 2) are maintained in a database residing in the Memory Manager. The Event Manager provides its management functions through its extended instruction set READ, TICKET, ADVANCE, and AWAIT, and in conjunction with the extended instructions TC_ADVANCE and TC_AWAIT provided by the Traffic Controller (to be discussed next). These extended instructions are based on the mechanism of eventcounts and sequencers [15]. The Event Manager verifies the access permission of every interprocess communication request through the Non-Discretionary Security Module. The extended instruction READ provides the current value of the eventcount requested by the caller. TICKET provides a complete time ordering of possibly concurrent events through the mechanism of sequencers. The Event Manager will be discussed in more detail in chapter IV.

3. Non-Discretionary Security Module

The purpose of the Non-Discretionary Security Module (NDS) is the enforcement of the non-discretionary security policy of the SASS. While the current implementation of SASS represents the Department of Defense security policy, any security policy which may be represented through a lattice structure [12] may also be implemented. The NDS is invoked via its extended instruction set: CLASS_EQ and CLASS_GE. The NDS is passed two classifications which it compares and then analyzes their relationship. CLASS_EQ will return a true value to the calling procedure only if the two classifications passed were equal. The CLASS_GE instruction will return true if a given classification is analyzed to be either greater than or equal to another given classification. The NDS does not utilize a data base as it works only with the parameters it is passed.

4. <u>Traffic Controller</u>

The task of processor scheduling is performed by the traffic controller. Saltzer [16] defines traffic controller as the processor multiplexing and control communication section of an operating system. The current SASS design utilizes Reed's [9] notion of a two level traffic controller, consisting of: 1) a Traffic Controller (TC) and 2) an Inner Traffic Controller (ITC).

The primary function of the Traffic Controller is the scheduling (binding) of user processes onto virtual processors. A virtual processor (VP) is an abstract data structure that simulates a physical processor through the preservation of an executing process' attributes (viz., the execution point and address space). Multiple VP's may exist for every physical processor in the system. Two VP's are permanently bound to Kernel processes (viz., Memory Manager and Idle) and as such are not in contention for process scheduling. These processes and their corresponding virtual processors are invisible to the TC. The remaining virtual processors are either idle or are temporarily bound to user processes as scheduled by the TC. The database utilized by the TC in process scheduling is the Active Process Table (APT). Figure 4 provides the structure of the APT.

The APT is a system-wide Kernel database containing an entry for every user process in the system. Since the current SASS design does not provide for dynamic process creation/deletion, a user process is active for the life of the system. Therefore, the size of the APT is fixed at the time of system generation. The APT is logically composed of three parts: 1) an APT header, 2) the main body of the APT, and 3) a VP table. The APT header includes: 1) a Lock to provide for a mutual exclusion mechanism, 2) a Running List indexed by VP ID to identify the current process running on each VP, 3) a Ready List, which points to the linked list of processes which are ready for scheduling, and 4) a Blocked List, which points to the linked list of processes which are in the blocked state awaiting the occurrence of some event.

A design decision was made to incorporate a single list of blocked processes instead of the more traditional

```
|---------------------------------|    ---|
| Lock                            |       |
|----------------|----------------|       |
| Running List   | APT Entry #    |       |
|                |----------------|       |
| VP ID----|     |----------------|       |
|          |     |----------------|       |
|          v     |----------------|       |
|----------------|----------------|       |--- APT
| Ready List Head| APT Entry #    |       |    HEADER
|                |----------------|       |
| Log_CPU_No--|  |----------------|       |
|             |  |----------------|       |
|             v  |----------------|       |
|---------------------------------|       |
| Blocked List Head               |       |
|---------------------------------|    ---|

|---APT Entry #
|
|   |----|------|------|--------|-----|-----|--|--------------|
|   |    |      |      |        |     |     |  |Awaited Event |
|   |    |      |      |        |     |     |  |--------------|
|   | AP | DBR  |Access|Priority|State|Affi-|VP|Handle        |
|   |Link|Handle|Class |        |     |nity |ID|   Instance   |
|   |    |      |      |        |     |     |  |   |     Count|
|   |----|------|------|--------|-----|-----|--|---|-----|----|
|   |    |      |      |        |     |     |  |   |     |    |
|   |----|------|------|--------|-----|-----|--|---|-----|----|
|   |    |      |      |        |     |     |  |   |     |    |
|   |----|------|------|--------|-----|-----|--|---|-----|----|
|   |    |      |      |        |     |     |  |   |     |    |
v   |----|------|------|--------|-----|-----|--|---|-----|----|
    |    |      |      |        |     |     |  |   |     |    |
    |----|------|------|--------|-----|-----|--|---|-----|----|

      Log_CPU_No----------->
     |----------|-----|-----|-----|-----|   ---|
     |NR_OF_VP'S|     |     |     |     |      |   TC
     |----------|-----|-----|-----|-----|      |--VP
     |FIRST_VP  |     |     |     |     |      |  TABLE
     |----------|-----|-----|-----|-----|   ---|
```

Figure 4. Active Process Table (APT)

notion of separate lists per eventcount because of its simplicity and its ease of implementation. This decision does not appreciably affect system performance or efficiency as the "blocked" list will never be very long. The VP table is indexed by logical CPU number and specifies the number of VP's associated with the logical CPU and its first VP in the Running List. The logical CPU number, obtained during system initialization, provides a simple means of uniquely identifying each physical CPU in the system. The main body of the APT contains the user process data required for its efficient control and scheduling. NEXT_AP provides the linked list threading mechanism for process entries. The DBR entry is a handle identifying the process' Descriptor Segment which is employed in process switching and memory management. The ACCESS_CLASS entry provides every process with a security label that is utilized by the Event Manager and the Segment Manager in the enforcement of the Non-Discretionary Security Policy. The PRIORITY and STATE entries are the primary data used by the Traffic Controller to effect process scheduling. AFFINITY identifies the logical CPU which is associated with the process. VP ID is utilized to identify the virtual processor that is currently bound to the process. Finally, the EVENTCOUNT entries are utilized by the TC to manage processes which are blocked and awaiting the occurrence of some event. HANDLE identifies the segment associated with the event, INSTANCE specifies the event, and COUNT determines which occurrence of the event is needed.

The Traffic Controller determines the scheduling order by process priority. Every process is assigned a priority at the time of its creation. Once scheduled, a process will run on its VP until it either blocks itself or it is preempted by a higher priority process. To insure that the TC will always have a process available for scheduling, there logically exists an "idle" process for every VP visible to the TC. These "idle" processes exist at the lowest process priority and, consequently, are scheduled only if there exists no useful work to be performed.

The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt or through its extended instruction set: ADVANCE, AWAIT, PROCESS_CLASS, and GET_DBR_NUMBER. ADVANCE and AWAIT are used to implement the IPC mechanism envoked by the Supervisor. PROCESS_CLASS and GET_DBR_NUMBER are called by the Segment Manager to ascertain the security label and DBR handle, respectively, of a named process. A more detailed discussion of the TC is provided in chapters III and IV.

5. Inner Traffic Controller

The Inner Traffic Controller is the second part of our two-level traffic controller. Basically, the ITC performs two functions. It multiplexes virtual processors onto the actual physical processors, and it provides the primitives for which inter-VP communication within the Kernel is implemented. A design choice was made to provide each physical processor in the system with a small fixed set of virtual processors. Two of these VP's are permanently bound to the Kernel processes. The Memory Manager is bound to the highest priority VP. Conversely, the Idle Process is bound to the lowest priority VP and, as a result, will only be scheduled if there exists no useful work for the CPU to perform. The primary database utilized by the ITC is the Virtual Processor Table (VPT). Figure 5 illustrates the VPT.

The VPT is a system wide Kernel database containing entries for every CPU in the system. The VPT is logically composed of four parts: 1) a header, 2) a VP data table, 3) a message table, and 4) an external VP list. The header includes a LOCK (spin lock) that provides a mutual exclusion mechanism for table access, a RUNNING LIST (indexed by logical CPU #) that identifies the VP currently running on the corresponding physical CPU, a READY LIST (indexed by logical CPU #) which points to the linked list of VP's which are in the "ready" state and awaiting scheduling on that CPU, and a FREE LIST which points to the linked list of unused entries in the message table. The VP data table contains the descriptive data required by the ITC to effectively manage the virtual processors. The DBR entry points within the MMU Image to the descriptor segment for the process currently running on the VP. PRI (Priority),

```
                        |-------------|-----------|      ---|
                        | Lock        |           |         |
                        |-------------|-----------|         |
                        | Running_List|VPT Entry #|         |
                        |             |-----------|         |
                        | CPU_No--|   |-----------|         |
                        |         |   |-----------|         |
                        |         V   |-----------|         |
                        |-------------|-----------|         |---VPT
                        | Ready_List  |VPT Entry #|         |  Header
                        |             |-----------|         |
                        | CPU_No--|   |-----------|         |
                        |         |   |-----------|         |
                        |         V   |-----------|         |
                        |-------------|-----------|         |
                        | Free_List   |           |         |
                        |-------------|-----------|      ---|

   |-----VP_ID
   |  |-----|---|-----|----|-------|---------|---|---|----|
   |  |NEXT |   |     |    |       |         |   |EXT|    |
   |  |READY|DBR|STATE|IDLE|VIRTUAL|PHYSICAL |PRI|VP |MSG |
   |  | VP  |   |     |FLAG|PREEMPT|PROCESSOR|   |ID |LIST|
   |  |-----|---|-----|----|-------|---------|---|---|----|
   |  |     |   |     |    |       |         |   |   |    |
   |  |-----|---|-----|----|-------|---------|---|---|----|
   V  |     |   |     |    |       |         |   |   |    |
      |-----|---|-----|----|-------|---------|---|---|----|
      |     |   |     |    |       |         |   |   |    |
      |-----|---|-----|----|-------|---------|---|---|----|
      |     |   |     |    |       |         |   |   |    |
      |-----|---|-----|----|-------|---------|---|---|----|

   |------MSG_INDEX
   |  |---------|-------|-----|      EXT_VP_ID--------->
   |  | NEXT_MSG| SENDER| MSG |     |-----------------------|
   |  |---------|-------|-----|     | VPT |    |    |    |  |
   |  |         |       |     |     |Entry|    |    |    |  |
   |  |---------|-------|-----|     | No  |    |    |    |  |
   |  |         |       |     |     |-----------------------|
   V  |---------|-------|-----|     |                       |
      |         |       |     |     |-----------------------|
      |---------|-------|-----|                 |
                                             External
      |-----------|-----------|                 VP
                  |                            List
            Message List
```

Figure 5. Virtual Processor Table (VPT)

STATE, IDLE_FLAG, and PREEMPT are the primary data used by the ITC for VP scheduling. PREEMPT indicates whether or not a virtual preempt is pending for the VP. The IDLE_FLAG is set whenever the TC has bound an "idle" process to the VP. Normally, a VP with the IDLE_FLAG set will not be scheduled by the ITC as it has no useful work to perform. In fact, such a VP will only be scheduled if the PREEMPT flag is set. This scheduling will allow the VP to be given (bound) to another process. PHYSICAL PROCESSOR contains an entry from the Processor Data Segment (PRDS) that identifies the physical processor that the VP is executing on. EXT_VP_ID is the identifier by which the VP is known by the Traffic Controller. A design choice was made to have the EXT_VP_ID equate to an offset into the External VP List. The External VP List specifies the actual VP ID (viz., VPT entry number) for each external VP identifier. This precluded the necessity for run time calculation of offsets for the EXT_VP_ID. NEXT READY VP provides the threading mechanism for the "Ready" linked list, and MSG LIST points to the first entry in the Message Table containing a message for that VP. The Message Table provides storage for the messages generated in the course of Inter-Virtual Processor communications. MSG contains the actual communication being passed, while SENDER identifies the VP which initiated the communication. NEXT_MSG provides a threading mechanism for multiple messages pending for a single VP.

The ITC is invoked by means of its extended instruction set: WAIT, SIGNAL, SWAP_VDBR, IDLE, SET_PREEMPT, and RUNNING_VP. WAIT and SIGNAL are the primitives employed in implementing the Inter-VP communication. SWAP_VDBR, IDLE, SET_PREEMPT, and RUNNING_VP are all invoked by the Traffic Controller. SWAP_VDBR provides the means by which a user process is temporarily bound to a virtual processor. IDLE binds the "Idle" process to a VP (the implication of this instruction will be discussed later). SET_PREEMPT provides the means of indicating that a virtual preempt interrupt is pending on a VP (specified by the TC) by setting the PREEMPT flag for that VP in the VPT. RUNNING_VP provides the TC with the external VP ID of the virtual processor currently running on the physical processor.

6. Distributed Memory Manager

The Distributed Memory Manager provides an interface structure between the Segment Manager and the Memory Manager Process. This interfacing is necessitated by the fact that the Memory Manager Process does not reside in the Distributed Kernel and consequently is not included in the user process' address space. The primary functions performed in this module are the establishment of Inter-VP Communication between the VP bound to its user process and the VP permanently bound to the Memory Manager Process, the manipulation of event data, and the dynamic allocation of available memory. The Distributed Memory Manager Module is invoked by the Segment Manager through its extended instruction set: MM_CREATE_ENTRY, MM_DELETE_ENTRY, MM_ACTIVATE, MM_DEACTIVATE, MM_SWAP_IN, and MM_SWAP_OUT. These extended instructions are utilized on a one to one basis by the extended instruction set of the Segment Manager (e.g., SM_SWAP_IN utilizes (calls) MM_SWAP_IN). Wells [6] provides a more detailed description of this portion of the Distributed Memory Manager and the extended instruction set associated with it.

The Distributed Memory Manager is also invoked through its remaining extended instructions: MM_READ_EVENTCOUNT, MM_TICKET, MM_ADVANCE, and MM_ALLOCATE. These Distributed Memory Manager functions will be discussed in detail in chapter IV.

E. NON-DISTRIBUTED KERNEL

The Non-Distributed Kernel is the second element residing in Level 1 of our abstract system view of the SASS. The sole component of the Non-Distributed Kernel is the Memory Manager Process.

1. Memory Manager Process

The primary purpose of the Memory Manager Process is the management of all memory resources within the SASS. These include the local and global main memories, as well as the hard-disk based secondary storage. A dedicated Memory Manager Process exists for every CPU in the system. Each CPU possesses a local memory where process local segments and shared, non-writeable segments are stored. There is also a global memory, to which every CPU has access, where the shared, writeable segments are stored. It is necessary to store these shared, writeable segments in the global memory to ensure that a current copy exists for every access.

The Memory Manager Process is tasked by other processes within the Kernel domain (via Signal and Wait) to perform memory management functions. These basic functions include the allocation/deallocation of local and global memory and of secondary storage, and the transfer of segments between the local and global memory and between secondary storage and the main memories. The extended instruction set provided by the Memory Manager Process includes: CREATE_ENTRY, DELETE_ENTRY, ACTIVATE, DEACTIVATE, SWAP_IN, and SWAP_OUT. These instructions correspond one to one with those of the Distributed Memory Manager Module. The system wide data bases utilized by all Memory Manager Processes are the Global Active Segment Table (G_AST), the Alias Table, the Disk Bit Map, and the Global Memory Bit Map. The processor local databases used by each Memory Manager Process are the Local Active Segment Table (L_AST), and the Local Memory Bit Map. Gary and Moore [4] provide a detailed description of the Memory Manager, its extended instruction set, and its databases.

A summary of the extended instruction set created by the components of the Security Kernel is provided by Figure 6. One might question the prudence of omitting PHYS_PREEMPT_HANDLER and VIRT_PREEMPT_HANDLER (viz., the handler routines for physical and virtual interrupts) from the extended instruction set as both of these interrupts may be raised (viz., initiated) from within the Kernel. A decision was made to not classify these handlers as "extended instructions" since they are only executed as the result of a physical or virtual interrupt and as such cannot be directly invoked (viz., "called") by any module in the system. A summary of the databases utilized by Kernel modules is presented in Figure 7.

F. SYSTEM HARDWARE

Level 0 of the SASS consists of the system hardware. This hardware includes: 1) the CPU, 2) the local memory, 3) the global memory, 4) the secondary storage (viz. hard disk), and 5) the I/O ports connecting the Host computer systems to the SASS. Since the SASS design allows for a multiprocessor environment, there may exist multiple CPU's and local memories. The target machine selected for the initial implementation of the system is the Zilog Z8001 microprocessor [17]. The Z8001 is a general purpose 16-bit, register oriented machine that has sixteen 16-bit general purpose registers. It can directly address 8M bytes of

MODULE	INSTRUCTION SET	
Segment Manager	Create_Segment*	Delete_Segment*
	Make_Known*	Terminate*
	SM_Swap_In*	SM_Swap_Out*
Event Manager	Read*	Ticket*
	Advance*	Await*
Non-Discretionary Security	Class_EQ	Class_GE
Traffic Controller	TC_Advance	TC_Await
	Process_Class	
Inner Traffic Controller	Signal	Wait
	Swap_VDBR	Idle
	Set_Preempt	Test_Preempt
	Running_VP	
Distributed Memory Manager	MM_Create_Entry	MM_Delete_Entry
	MM_Activate	MM_Deactivate
	MM_Swap_In	MM_Swap_Out
Non-Distributed Memory Manager	Create_Entry	Delete_Entry
	Activate	Deactivate
	Swap_In	Swap_Out

* Denotes user visible instructions

Figure 6. Extended Instruction Set

MODULE	DATABASE
Gate Keeper	Parameter Table
Segment Manager	Known_Segment_Table (KST)
Traffic Controller	Active_Process_Table (APT)
Inner Traffic Controller	Virtual_Processor_Table (VPT)
	Memory_Management_Unit Image (MMU)
Memory Manager	Global_Active_Segment_Table (G_AST)
	Local_Active_Segment_Table (L_AST)
	Disk_Bit_Map
	Global_Memory_Bit_Map
	Local_Memory_Bit_Map

Figure 7. Kernel Databases

memory, extensible to 48M bytes. The Z8001 architecture supports memory segmentation and two-domain operations. The memory segmentation capability is provided externally by the Zilog Z8010 Memory Management Unit (MMU). The Z8010 MMU [18] provides management of the Z8001 addressable memory, dynamic segment relocation, and memory protection. Memory segments are variable in size from 256 bytes to 64K bytes and are identified by a set of 64 Segment Descriptor Registers, which supply the information needed to map logical memory addresses to physcal memory addresses. Each of the 64 Descriptor Registers contains a 16-bit base address field, an 8-bit limit field, and an 8-bit attribute field. Unfortunately, the Z8001 hardware was not available for use during system development. Therefore, all work to date has been completed through utilization of the Z8002 non-segmented version of the Z8000 microprocessor family [17]. The actual hardware used in this implementation is the Advanced Micro Computers Am96/4116 MonoBoard Computer [19] containing the AmZ8002 sixteen bit non-segmented microprocessor. This computer provides 32K bytes of on-board RAM, 8k bytes of PROM/ROM space, two RS232 serial I/O ports, 24 parallel I/O lines, and a standard INTEL Multibus interface. The general structure of the design has been preserved by simulation of the segmentation hardware in software. This software MMU Image (see Figure 8) is created as a database within the Inner Traffic Controller.

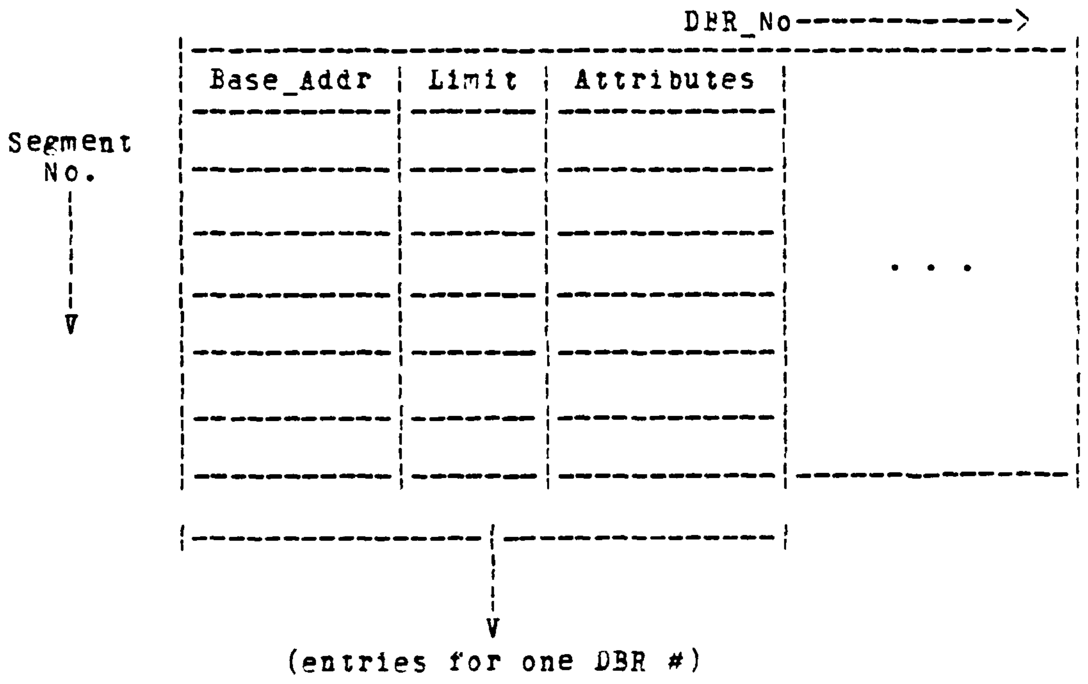

Figure 8. Memory Management Unit (MMU) Image

The MMU Image is a processor-local database indexed by DBR_No. Each DBR_No represents one record within the MMU Image. Each record is an exact software copy of the Segment Descriptor Register set in the hardware MMU. Each element of this software MMU Image is in the same form utilized by the special I/O instructions to load the hardware MMU. Each DBR record is indexed by segment number (Segment_No). Each Segment_No entry is composed of three fields: Base_Addr, Limit, and Attributes. Base_Addr is a 16-bit field which contains the base address of the segment in physcal memory. Limit is an 8-bit field that specifies the number of contiguous blocks of memory occupied by the segment. Attributes is an 8-bit field representing the eight flags which specify the segment's attributes (e.g., "read", "execute", "write", etc.).

G. SUMMARY

An extended overview of the current SASS design has been presented in this chapter. The four major levels of abstraction comprising the SASS system have been identified and the major components of each level have been discussed. The extended instruction set provided by the SASS Kernel was also defined. With this background, the actual details of this implementation will be described in chapters III and IV.

III. IMPLEMENTATION ISSUES

Issues bearing on the implementation of process management and refinements made to existing modules are presented in this chapter. Process management for the SASS was provided through the implementation of the Traffic Controller Module, the Event Manager Module, the Distributed Memory Manager Module, and a Gate Keeper Stub (system trap). Additionally, since a demonstration/testbed was integral to the testing and verification of the implementation, it was necessary to complete other supportive tasks. These supportive tasks included limited Kernel database initialization, revised preempt interrupt handling mechanisms, Idle process definition and structure, and additional refinements to existing modules.

A. DATABASE INITIALIZATION

Previous work on SASS has relied on statically built databases, which proved to be sufficient for demonstration of a single processor, single host supported system. In the current demonstration, multiple hosts are simulated, and the Kernel data structures have been refined to represent a multiprocessor environment. Since a multiprocessor system was unavailable at the time of this demonstration, several "runs" were made and traced, using different logical CPU numbers, to show the correctness of this structure. Due to this multiprocessor representation and simulation of multiple hosts, the use of statically built Kernel databases was no longer convenient. Therefore, it became necessary to provide initialization routines for the dynamic creation of those Kernel databases required for this implementation. While it was not the intent of this effort to implement system initialization, care was taken in the writing of these initializing routines so that they might be utilized in the system intialization implementation with, hopefully, minimal refinement. Database initialization was restricted to those databases existing in the Inner Traffic Controller and the Traffic Controller. Limited elements of the Known Segment Table (KST) and Global Active Segment Table (G_AST) were also created for demonstration purposes.

1. Inner Traffic Controller Initialization

A "Bootstrap Loader" Module, which logically exists at a higher level of abstraction within the Kernel, was created to initialize the databases of the Inner Traffic Controller. This initialization includes the creation of: 1) the Processor Data Segment (PRDS), 2) an MMU Map, 3) Kernel domain stack segments for Kernel processes, 4) allocation and updating of MMU entries for Kernel processes, and 5) Virtual Processor Table (VPT) entries.

The PRDS was loaded with constant values that specify the physical CPU ID, logical CPU ID, and number of

VP's allocated to the CPU. A design decision was made to allocate logical CPU ID's in increments of two (beginning with zero) so that they could be used to directly access lists indexed by CPU number. The MMU map, constructed as a "byte" map, was created to specify allocated and free MMU Image entries.

A separate procedure, CREATE_STACK, was created to establish the initial Kernel domain stack conditions for Kernel processes. A discussion and diagram of these initial stack conditions is presented in the next section. ALLOCATE_MMU checks the MMU Map and allocates the next availabe MMU entry to the process being created. The PRDS is inserted in the allocated MMU entry and the DBR number is returned to the calling procedure. The DBR number (handle) is merely the offset of the DBR in the MMU Image. Since the ITC deals with an address rather than a handle, a procedure, GET_DBR_ADDR, was created to convert this offset into a physical address. UPDATE_MMU_IMAGE is the procedure which creates or modifies MMU Image entries. UPDATE_MMU_IMAGE accepts as arguments the DBR number, segment number, segment attributes, and segment limits. To facilitate process switching and control, various process segments must possess the same segment number system wide. This is accomplished during initialization through the use of the UPDATE_MMU_IMAGE procedure. In the ITC, these segments include the PRDS (segment number zero) and the Kernel stack segment (segment number one).

The final task required in ITC intialization is the creation of the VPT. The VPT header is initialized with the "running" and "ready" lists pointers set to a 'nil' state, and the "free" list pointer set to the first entry in the message table. Virtual Processor entries are inserted in the main body of the VPT by the UPDATE_VP_TABLE procedure. Entries are first made for the VP's permanently bound to the Memory Manager and Idle processes. The VP bound to the MM process is given a priority of 2 (highest), and the VP bound to the Idle process is given a priority of 0 (lowest). The External VP ID for both of these VP's is set to "nil" as they are not visible to the Traffic Controller. The remaining VP's allocated to the CPU (viz., TC visible VP's) are then entered in the VPT with a priority of 1 (intermediate), and their "idle" and "preempt" flags are set. The preempt flag is set for these TC visible VP's to insure proper scheduling by the Traffic Controller. The DBR for these remaining VP's is initialized with the Idle process DBR. A discussion of "idle" processes and VP's will be provided later in this chapter. The External VP ID for each TC visible VP is merely the offset of the next available entry in the EXTERNAL VP LIST. This External VP ID is entered in the VPT, and the corresponding VP ID (viz., VPT Entry #) is entered in the EXTERNAL VP LIST.

Once these VPT entries have been made, it is necessary to set the state of each VP to "ready" and thread them (by priority) into the appropriate ready list. A VPT threading mechanism was provided by Reitz [5] in procedure MAKE_READY. However, it was desired to have a more general threading mechanism that could be used for other lists. Procedure LIST_INSERT was created to provide this general threading mechanism. LIST_INSERT is logically a "library" function that exists at the lowest level of abstraction in the Kernel. This function threads an object into a list (specified by the caller) in order of priority, and then sets its state as specified by the calling parameters.

Once the "Bootstrap Loader" has completed ITC initialization, it passes control to the ITC GETWORK procedure to begin VP scheduling.

2. Traffic Controller Initialization

The initialization routines for the TC include TC_INIT, CREATE_PROCESS, and CREATE_KST. These routines are called from the Memory Manager process. The MM process was chosen to initiate these routines as it is bound to the highest priority VP and will begin running immediately after the Inner Traffic Controller is initialized. Procedure MM_ALLOCATE was written to allocate memory space for data structures during initialization (viz., Kernel stacks, user stacks, and KST's). Memory space is allocated in blocks of 100 (hex) bytes. MM_ALLOCATE is merely a stub of the memory allocating procedure designed by Moore and Gary [4].

It was necessary to pass long lists of arguments to the TC for initialization purposes. To aid in this passing of parameters, a data structure template was used. This template was created by declaring the parameters as a data structure in both the sending and receiving procedures, and then imaging this structure at absolute address zero. The process' stack pointer was then decremented by the size of the parameter data structure, and the parameters were loaded into this data structure indexed by the stack pointer. This template made it very easy to send and receive long argument lists using the process' stack segment.

TC_INIT initializes the APT header and virtual interrupt vector (discussed later). Each element of the running list is marked "idle", the ready and blocked lists are set to "nil", and the number of VP's and first VP for each CPU are entered in the VP table. The address of the virtual preempt handler is then passed to the ITC procedure CREATE_INT_VEC for insertion in the virtual interrupt vector.

CREATE_PROCESS intializes user processes and creates entries in the APT. ALLOCATE_MMU is called to acquire a DBR number, and an APT entry is created with the process descriptors (viz., parameters). The process is then declared "ready" and threaded into the approciate ready list by calling the threading function, LIST_INSERT. A user stack is allocated and UPDATE_MMU_IMAGE is called to include the user stack in the MMU as segment number three. The user stack contains no information or user process initialization parameters (viz., execution point and address space) as all processes are initialized and begin execution from the Kernel domain. Next, a Kernel domain stack is allocated and included in the MMU Image. A design decision was made to initialize the Kernel stacks for user processes with the same structure as the Kernel process' stacks. The rationale for this decision is presented in the next section. As a result of this decision, it became possible to use the CREATE_STACK procedure in building Kernel domain stacks for both Kernel and user prosesses. CREATE_STACK was therefore used as a library function and placed in the library module with LIST-INSERT.

Finally, a Known Segment Table (KST) stub is created to provide a means of demonstrating the mechanism provided by the eventcounts and sequencers for interprocess communication (IPC) and mutual exclusion. Space for the process' KST is created by calling MM_ALLOCATE. The KST is then included in the process' address space, as segment number two, by UPDATE_MMU_IMAGE. Initial entries are made in the Known Segment Table by procedure CREATE_KST. CREATE_KST makes an entry in the KST for the "root" and marks the remaining KST entries as "available." The Unique_ID portion of the root's handle (viz., upper two words) is initialized as -1 (for convenience) and the G_AST entry number portion of the handle (viz., lowest word) is initialized with zero.

3. Additional Initialization Requirements

As already mentioned, the Memory Manager Process prepares the arguments utilized by TC_INIT, CREATE_PROCESS, and CREATE_KST for TC initialization and user process creation. Additionally, the MM process creates a Global Active Segment Table (G_AST) stub utilized for demonstration of event data management. The G_AST stub is declared in a separate module (viz., the DEMO_DATABASE Module) with the format prescribed by Moore and Gary [4]. However, the only fields initialized and utilized by this implementation are UNIQUE_ID, SEQUENCER, INSTANCE 1, and INSTANCE 2. The eventcounts and sequencer fields are initialized as zero whenever an entry is created in the G_AST. The UNIQUE_ID is created just to support this demonstration and does not reflect the segment's unique identifier as specified by Moore and Gary [4]. In this demonstration, UNIQUE_ID is built with the parameters passed to MM_ACTIVATE. The first word in UNIQUE_ID is the G_AST entry number of the segment's parent, and the second word is the segment's entry number into the alias table. The UNIQUE_ID together with the offset of the segment's entry in the G_AST comprise the segment HANDLE maintained in the KST. The first entry in the G_AST is reserved for the root, and is initialized with an Unique_ID of minus one (-1). It should be noted that any call to MM_ACTIVATE for a segment already possessing an entry in the G_AST will not effect any changes to that entry. This is to insure that a single G_AST entry exists for every segment as specified by Moore and Gary [4].

B. PREEMPT INTERRUPTS

Various refinements were made in the handling of both physical (hardware) and virtual (software) preempt interrupts. A hardware preempt is a non-vectored interrupt that invokes the virtual processor scheduling mechanism (viz., ITC GETWORK). A virtual preempt is a software vectored interrupt that invokes the user process scheduling mechanism (viz., TC_GETWORK). This implementation provides the notion of a virtual interrupt that closely mirrors the behavior of a hardware interrupt. In particular, there are similar constructs for initialization of a handler, invokation of a handler, masking of interrupts, and return from a handler. As with most hardware interrupts, a virtual interrupt can occur only at the completion of execution for an "instruction," where each kernel entry and exit for a process delimit a single "virtual instruction."

1. Physical Preempt Handler

The physical preempt handler resides in the virtual processor manager (viz., Inner Traffic Controller). The functions it perform are: 1) save the execution point, 2) invoke ITC GETWORK, 3) check for virtual preempt interrupts, 4) restore the execution point, and 5) return control via the IRET instruction. Reitz [5] included the hardware preempt handler in ITC GETWORK by establishing two entry points and two exit points, one for a regular call to GETWORK and another for the preempt interrupt. He had a separate procedure, TEST_PREEMPT, that was used to check for the occurrence of virtual preempt interrupts. This structure works nicely, but it requires some means of determining how GETWORK was invoked so that the proper exiting mechanism is used. This was resolved by incorporating a preempt interrupt flag in the status register block of every process' Kernel domain stack segment. A design decision was made to restructure the hardware preempt handler into a single and separate procedure, PHYS_PREEMPT_HANDLER. This allowed ITC GETWORK to have a single entry and exit point, and it did away with the necessity of maintaining a preempt interrupt flag in the process stacks. PHYS_PREEMPT_HANDLER was constructed from the preempt handling code in GETWORK and procedure TEST_PREEMPT. TEST_PREEMPT was deleted from the ITC as its functions were performed by PHYS_PREEMPT-HANDLER.

A further refinement was made to the hardware preempt handler dealing with the method by which the virtual preempt handler was invoked. Reitz [5] invoked the virtual preempt handler from TEST_PREEMPT by means of the "call" instruction. Since the virtual preempt handler logically exists at a higher level of abstraction than the ITC, this invocation violated our notion of only allowing "calls" to lower or equal abstraction levels. However, this deviation was necessitated by the absence of a virtual interrupt structure. This problem was alleviated by creating a virtual interrupt vector in the ITC that is used in the same way as the hardware interrupt vector. The virtual preempt was given a virtual interrupt number (zero). The virtual interrupt handler is then invoked by means of a "jump" through the virtual interrupt vector for virtual interrupt number 0. This invocation occurs in the same manner that the handlers for hardware interrupts are invoked. The virtual interrupt vector is created by procedure CREATE_INT_VEC. CREATE_INT_VEC accepts as arguments a virtual interrupt number and the address of the interrupt handler. The creation of the virtual preempt entry in the virtual interrupt vector is accomplished at the time of the Traffic Controller initialization by TC_INIT.

2. Virtual Preempt Handler

The virtual preempt handler (VIRT_PREEMPT_HANDLER) resides in the user process manager (viz., the Traffic Controller). The functions performed by VIRT_PREEMPT_HANDLER are: 1) determine the VP ID of the virtual processor being preempted, 2) invoke the process scheduling mechanism (viz., TC_GETWORK), and 3) return control via a virtual interrupt return. The correct VP ID is obtained by calling RUNNING_VP in the ITC. The Active Process Table is then locked, and the state of the process running on that VP is changed to "ready." At this time, process scheduling is effected by calling TC_GETWORK. Once process scheduling is completed, the APT is unlocked and control is returned via a virtual interrupt return. This virtual interrupt return is merely a jump to the PREEMPT_RET label in the hardware preempt handler (This jump emulates the action of the IRET instruction for a hardware interrupt return). This label is the point at which the virtual preempt interrupts are unmasked.

All Kernel processes are initialized to appear as though they are returning from a hardware preempt interrupt. All user processes initially appear to be returning from a virtual preempt interrupt. Therefore, the initial conditions of a process' Kernel domain stack is largely influenced by the stack manipulation of the preempt handlers. Figure 9 illustrates the initial Kernel domain stack structure for all system processes.

The initial Kernel Flag Control Word (FCW) value is "5000", indicating non-segmented code, system mode of operation, non-vectored interrupts masked, and vectored interrupts enabled. The Current Stack Pointer value is set to the first entry in the stack (viz., SP). The IRET Frame is the portion of the Kernel stack affected by the IRET instruction. The first element, Interrupt ID (set to "FFFF") is merely popped off of the stack and discarded. The next element, Initial FCW, is popped and placed in the system Flag Control Word. Initial FCW is set to "5000" for Kernel

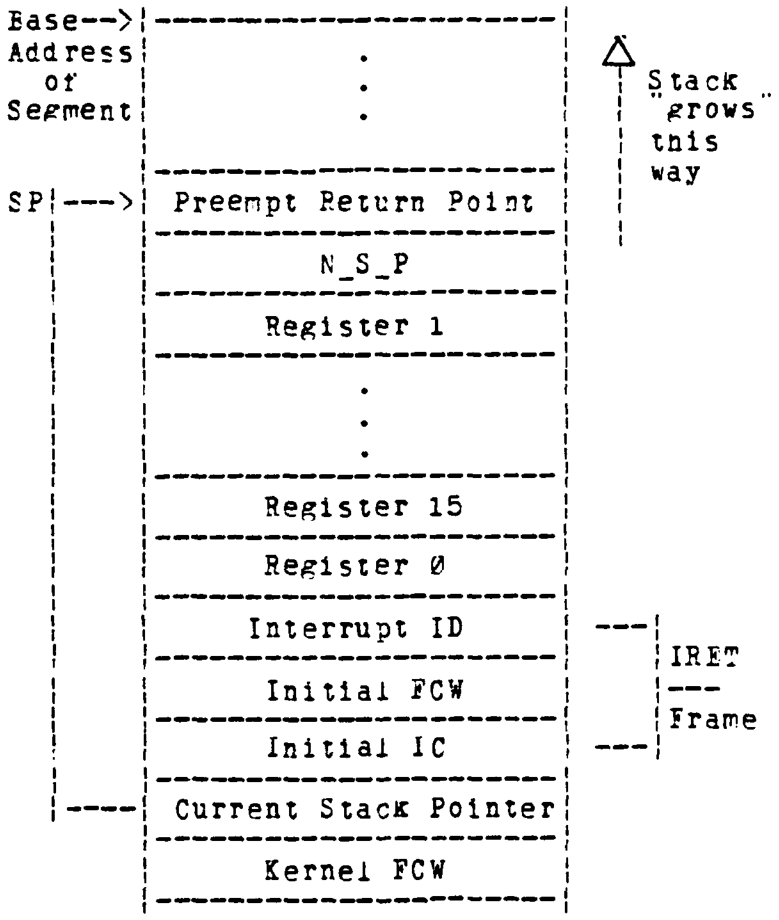

Figure 9. Initial Process Stack

processes and "1800" (indicating normal mode with all interrupts enabled) for user processes. The final element of the IRET frame, Initial IC is popped off of the stack and placed in the program counter (PC) register. This value is initialized as the entry address of the process in question.

The "register" entries on the stack represent the initial register contents for the process at the beginning of its execution. Since the Kernel processes (viz., MM and Idle) do not require any specific initial register states, their entries reflect the register contents at the time of stack creation. Initial register conditions are used to provide initial "parameters" required by the user processes. This will depend largely upon the parameter passing conventions of the implementation language. The means for register initialization was provided through CREATE_PROCESS; however, the only initial register condition used for the user processes in this demonstration was register #13. Register #13 was used to pass the user ID/Host number of the process created. This value is utilized by the user process in activating the segment used for inter-process communication between a Host's File manager and I/O processes. Another logical parameter passed to the user processes is the root segment number. This did not require a register for passing in the demonstration as it is known to be the first entry in the KST for all processes. The N_S_P entry on the stack represents the initial value of the normal stack pointer. For user processes, this value is obtained when the Supervisor domain stack for that process is created. For Kernel processes, this value is set to "FFFF" since they execute solely in the Kernel domain and have no Superivsor domain stack. The Preempt Return Point specifies the address where control will be passed once the process' VP is scheduled and the "return" from ITC GETWORK is executed. For Kernel processes, this is the point within the hardware preempt handler where the virtual processor table is unlocked. For user processes, this is the point within the virtual preempt handler where the Active Process Table is unlocked.

It is important to note that if the APT was not unlocked when a user process began its initial execution, the system would become deadlocked and no further process scheduling could occur. It should be further noted that the initial stack conditions for user processes do not reflect a valid history of execution. The "normal" history of a user process returning from ITC GETWORK after a virtual preempt interrupt would reflect the passing of control through SWAP_VDBR and TC_GETWORK to the point in the virtual preempt handler where the APT is unlocked. Another "possible" history could reflect the occurrence of a hardware preempt interrupt at the point in the virtual preempt handler where the APT is unlocked. Such a history would be depicted by replacing the current top of the stack with the return point into the hardware preempt handler (viz., at the point of virtual preempt interrupt unmasking) and an additional hardware preempt interrupt frame whose IC value in the IRET frame is the point in the virtual preempt handler where the APT is unlocked. The current initial stack condition for user processes was chosen for its ease of understanding and its clear depiction of the fact that the structure of a Kernel domain stack is the same for both Kernel and user processes.

C. IDLE PROCESSES

In the SASS design, there logically exists a Kernel domain "Idle" process for every physical processor in the system and a Supervisor domain "Idle" process for every "TC visible" virtual processor in the system. These processes are necessary to insure that both the VP scheduler (viz., ITC GETWORK) and the process scheduler (TC_GETWORK) will always have some object to schedule, hence precluding any CPU or VP from ever having an undefined execution point. Since the Kernel domain Idle process performs no useful work, it could be included within the ITC by means of an infinite looping mechanism. The Kernel Idle process was maintained separately, however, as it is hoped that future work on SASS will provide this Idle process with some constructive purpose (e.g., performing maintenance diagnostics).

The Supervisor domain Idle processes (hereafter referred to as TC Idle processes) are scheduled (bound) on VP's when there are no user processes awaiting scheduling. Since a TC Idle process performs no user constructive work, we do not want any VP executing a TC Idle process to be bound to a physical processor. In other words, a VP bound to a TC Idle process assumes the lowest system priority (represented by the "idle flag"). Therefore, any such VP will have its idle flag set and will not be scheduled unless it receives a virtual preempt interrupt. Such an interrupt will allow the VP to be rescheduled by the Traffic Controller. It should be obvious, at this point, that a TC Idle process will never actually begin execution on a real processor. This knowledge allowed a design decision to be made to only simulate the existence of TC Idle processes. At the TC level, this was accomplished by a constant value, IDLE_PROC, that was used as a process ID in the APT running list, thus precluding the necessity of any "Idle" entries in the APT. At the ITC level, any VP marked "Idle" (viz., the idle flag set) was given the DBR number (viz., address space) of the Kernel Idle process solely to provide the use of a Kernel domain stack for rescheduling of the VP.

D. ADDITIONAL KERNEL REFINEMENTS

In addition to those already discussed, several other refinements to existing Kernel modules were effected in this implementation. One of these refinements deals with the way virtual processors are identified by the Traffic Controller. In the current implementation, all TC visible virtual processors are given an External VP ID which corresponds to its entry number in an External VP List. This required a modification to the ITC procedure RUNNING_VP. The benefits derived from this refinement included the ability to directly access the External VP ID in the Virtual Processor Table vice the requirement of a run time division to compute its value and the ability to use the External VP ID as an index into the TC running list.

Refinements were also made to the existing Memory Manager, File Manager, and IO process stubs used for demonstration purposes. These refinements were largely associated with the eventcount and sequencer mechanisms utilized in this implementation. The current status of these processes is provided in a report by Schell and Cox [22].

The remaining refinements deal largely with the MMU Image. In Moore and Gary's [4] design, the MMU Image was managed by the Memory Manager process. This was largely because the MMU Image is a processor local database and would seem well suited for management by the non-distributed Kernel. In fact, the MMU Image is utilized mainly by the ITC for the multiplexing of process address spaces. Therefore, in the current design, the MMU I[illegible]s are maintained by the Inner Traffic Controller. However, the MMU header proposed by Moore and Gary (viz., the BLOCKS_USED and MAXIMUM_AVAILABLE_BLOCKS fields) was retained in the Memory Manager as it is used strictly in the management of a process' virtual core and is not associated with the hardware MMU.

In Wells' design [6], the Traffic Controller used the linear ordering of the DBR entries in the MMU Image as the DBR handle (viz., 1,2,3...). This required a run time division operation to compute the DBR number, and a run time multiplication operation, by MM_GET_DBR_VALUE, to recompute the DBR address for use by the ITC. In the current design, the offset of the DBR entry in the MMU Image (obtained at the time of MMU allocation) is used as the DBR handle in the Traffic Controller. Furthermore, SWAP_VDBR was refined to accept a DBR handle rather than a DBR address to preclude the necessity of the Traffic Controller having to deal with MMU addresses. DBR addresses are computed only within the ITC (viz., by procedure GET_DBR_ADDR) by adding the value of the DBR handle to the base address of the MMU Image. Since DBR addresses are now used solely within the ITC, procedure MM_GET_DBR_VALUE was no longer needed and was deleted from the Memory Manager.

E. SUMMARY

The primary issues addressed in this thesis effort have been presented in this chapter. Aside from the process management functions, this description included a mechanism for limited Kernel database initialization, a revised preempt interrupt handling mechanism, the creation of a virtual interrupt structure, a definition of "idle" processes and their structure, and a discussion of the minor refinements effected in existing SASS modules. A detailed description of the implementation of process management functions for the SASS is presented in the next chapter.

IV. PROCESS MANAGEMENT IMPLEMENTATION

The implementation of process management functions and a gate keeper stub (system trap) is presented in this chapter. The implementation is discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, User Gate, and Kernel Gate Keeper modules. A block diagram depicting the structure and interrelationships of these modules is presented in figure 10. Support in developing the Z8000 machine code for this implementation was provided by a Zilog MCZ Developmental System operating under the RIO operating system. The Developmental System provided disk file management for a dual drive, hard sectored floppy disk, a line oriented text editor, a PLZ/ASM assembler, a linker and a loader that created an executable image of each Z8000 load module. An upload/download capability with the Am96/4116 MonoBoard computer was also provided. This capability, along with the general interfacing of the Am96/4116 into the SASS system, was accomplished in a concurrent thesis endeavor by Gary Baker. Baker's work relating to hardware initialization in SASS, will be published upon completion of his thesis work in June 1981.

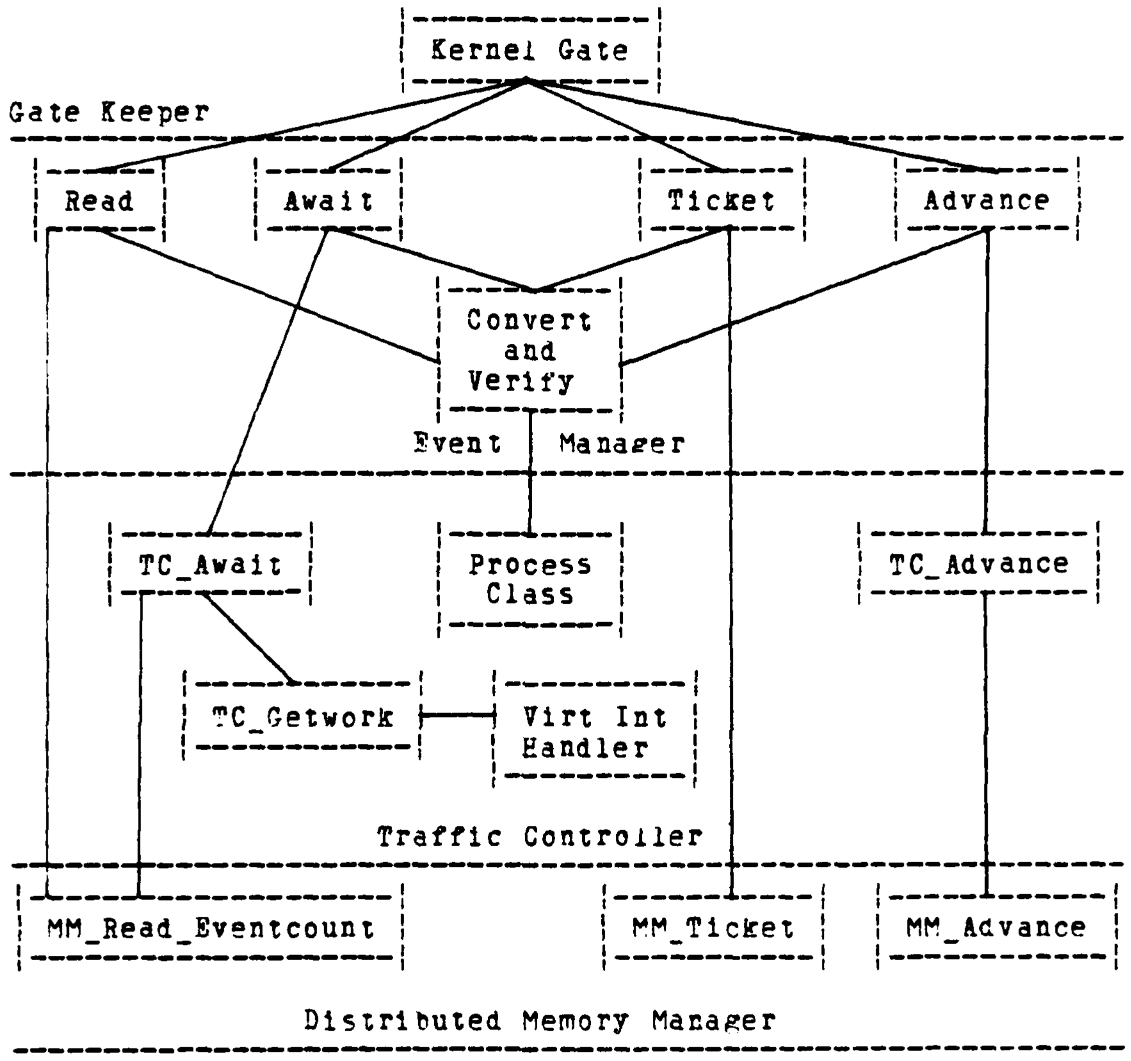

Figure 10. Implementation Module Structure

A. EVENT MANAGER MODULE

The eventcount and sequencer primitives [15], which are system-wide objects, collectively comprise the event data of SASS. As mentioned earlier, this event data is tied directly to system segments and is stored in the Global Active Segment Table. There are two eventcounts and one sequencer for every segment in the system. These objects are identified to the Kernel in user calls by specification of a segment number. Once this segment number is identified by the Kernel, the segment's handle can be obtained from the process' Known Segment Table. The segment handle identifies the particular entry in the G_AST containing the event data desired.

The Event Manager module manages the event data within the system and provides the mechanism for interprocess communication between user processes. The Event Manager consists of six procedures. Four of these (Advance, Await, Read, and Ticket) represent the four user extended instructions provided by the Event Manager. The remaining two procedures provide internal computational support to include necessary security checking. The Event Manager is invoked solely by user processes, via the Gate Keeper, through utilization of the extended instruction set provided. For every Event Manager extended instruction invoked by a user process, the non-discretionary security is verified by comparing the security access classification of the process invoking the instruction with the classification of the object (segment) being accessed. Access to the user process' Known Segment Table is required by the module in order to ascertain the segment handle and security class for a given segment number. The PLZ/ASM assembly language listing for the Event Manager module is provided in Appendix A. A more detailed discussion of the procedures comprising the Event Manager follows.

1. Support Procedures

The procedures GET_HANDLE and CONVERT_AND_VERIFY provide internal support for the Event Manager and are not visible to the user processes. Procedure CONVERT_AND_VERIFY is invoked by the four procedures representing the instruction set of the Event Manager. The input parameters to CONVERT_AND_VERIFY are a segment number and a requested mode of access (viz., read or write). CONVERT_AND_VERIFY returns a pointer to the segment's handle and a success code. Procedure GET_HANDLE is invoked solely by CONVERT_AND_VERIFY. The input parameter to GET_HANDLE is the segment number received as input by CONVERT_AND_VERIFY. GET_HANDLE returns a pointer to the segment's handle, a pointer to the segment's security classification, and a success code. A discussion of the functions provided by these support procedures follows.

Procedure GET_HANDLE translates the segment number, received as input, into a KST index number and verifies that the resulting index number is valid. Next the base address of the process' KST is obtained from procedure ITC_GET_SEG_PTR. The KST index number is then converted into a KST offset value and added to the base address to obtain the appropriate KST entry pointer for the segment in question. A verification is then made to insure that the referenced segment is "known" to the process. If the segment is not known, an error message is returned to CONVERT_AND_VERIFY. Otherwise, a pointer to the segment's handle is obtained to identify the segment to the memory manager. A pointer to the segment's security class entry in the KST is also returned for use in appropriate security checks.

Procedure CONVERT_AND_VERIFY provides the necessary non-discretionary security verification for the extended instruction set of the Event Manager. Procedure GET_HANDLE is invoked for segment number verification and to obtain pointers to the segment's handle and security class. If GET_HANDLE returns with a successful verification, the process' security class is compared to the segment's security class to verify the mode of access requested. A request for "write" access causes invocation of the CLASS_EQ function in the Non-Discretionary Security Module to insure that the security classification of the process is equal to the classification of the eventcount or sequencer, which is the same as that of the segment. Otherwise, the CLASS_GE function is called to verify that the process has read access. If the appropriate security check is unsuccessful, an error code is returned by CONVERT_AND_VERIFY. Otherwise, the segment handle is returned along with a success code of "succeeded" indicating that the user process possesses the necessary security clearance to complete execution of the extended instruction.

2. <u>Read</u>

Procedure READ ascertains the current value of a user specified eventcount and returns its value to the caller. The input parameters to READ are a segment number and an instance (viz., an event number). CONVERT_AND_VERIFY is invoked with a "read" access request to obtain the segment's handle and necessary verification. "Read" access is sufficient for this operation as it only requires observation of the current eventcount value and performs no data modification. If verification is successful, procedure MM_READ_EVENTCOUNT is called to obtain the eventcount value.

3. <u>Ticket</u>

Procedure TICKET returns the current sequencer value for the segment specified by the user. CONVERT_AND_VERIFY is called with a request for write access to obtain verification and the segment handle. Write access is required because once the sequencer value is read it must be incremented in anticipation of the next ticket request. Once verification is complete, MM_TICKET is invoked to obtain the sequencer value that is returned to the user process. It is noted that every call to TICKET for a particular segment number will return a unique and time ordered sequencer value. This is because the sequencer value may only be read within MM_TICKET while the G_AST is locked, thereby preventing simultaneous read operations. Futhermore, once the sequencer value is read it is incremented before the G_AST is unlocked.

4. Await

Procedure AWAIT allows a user process to block itself until some specified event has occurred. The parameters to AWAIT include a segment number and instance, which identify a particular event, and a user specified value which identifies a particular occurrence of the event. Verification of read access and a pointer to the segment's handle is obtained from procedure CONVERT_AND_VERIFY. Procedure TC_AWAIT is invoked to effect the actual waiting for the event occurrence. TC_AWAIT will not return to AWAIT until the requested event has occurred. It is noted that AWAIT makes no assumptions about the event value specified by the user. Therefore, the Kernel cannot guarantee that the event specified by the user will ever occur; this is the responsibility of other cooperating user processes.

5. Advance

Procedure ADVANCE allows a user process to broadcast the occurrence of some event. This is accomplished by incrementing the value of the eventcount associated with the event that has occurred. The parameters to ADVANCE include a segment number and instance which identify a particular event. The calling process must have write access to the identified segment as modification of the eventcount is required. Verification of write access and a pointer to the segment's handle is obtained through procedure CONVERT_AND_VERIFY. Procedure TC_ADVANCE is invoked to perform the actual broadcasting of event occurrence.

B. TRAFFIC CONTROLLER MODULE

The primary functions of the Traffic Controller module are user process scheduling and support of the inter-process communication mechanism. The Traffic Controller is invoked by the occurrence of a virtual preempt interrupt and by the Event Manager and the Segment Manager through the extended instruction set: TC_Advance, TC_Await, Process_Class, and Get_DBR_NUMBER. The Traffic Controller module is comprised of nine procedures. Four of these procedures represent the extended instruction set of the Traffic Controller. A detailed discussion of six of the procedures contained in the Traffic Controller module is presented below. The remaining three procedures (viz., TC_INIT, CREATE_PROCESS, and CREATE_KST) were described in chapter three. The PLZ/ASM assembly language source code listings for the Traffic Controller module is provided in Appendix B.

1. TC Getwork

Procedure TC_GETWORK provides the mechanism for user process scheduling. The input parameters to TC_GETWORK are the VP ID of the virtual processor to which a process will be scheduled and the logical CPU number to which the virtual processor belongs. The determination of which process to schedule is made by a looping mechanism that finds the first "ready" process on the ready list associated with the current logical CPU number. Processes appear in the ready list by order of priority. This looping mechanism is required as both "running" and "ready" processes are maintained on the ready list. This ready list structure was chosen to simplify the algorithm provided in procedure TC_Advance. If a ready process is found, its state is changed to "running" and its process ID (viz., the APT entry number) is inserted in the running list entry associated with the current virtual processor. Procedure SWAP_VDBR is then invoked in the Inner Traffic Controller to effect the actual process switch. If a ready process was not found (viz., the ready list was empty or comprised solely of "running processes"), then the running list entry associated with the current virtual processor is marked with the constant "Idle_Proc" and procedure IDLE is invoked in the Inner Traffic Controller.

2. TC Await

The primary function of TC_AWAIT is the determination of whethe. some user specified event has occurred. If the event has occured, control is returned to the caller. Otherwise, the process is blocked and another process is scheduled. The input parameters to TC_AWAIT are a pointer to a segment handle, an instance (event number), and a user specified eventcount value. TC_AWAIT initially locks the Active Process Table and obtains the current value of the eventcount in question by calling procedure MM_READ_EVENTCOUNT. The determination of event occurrence is made by comparing the user specified eventcount value with the current eventcount. If the user value is less than or equal to the current eventcount, the awaited event has occurred and control is returned to the caller. Otherwise, the awaited event has not yet occurred and the process must be blocked.

If the process is to be blocked, procedure RUNNING_VP is invoked to ascertain the VP ID of the virtual processor bound to the process. The process' ID (viz., APT entry number) is then read from the running list. The input parameters to TC_AWAIT (viz., Handle, Instance, and Value) are then stored in the Event Data portion of the process' APT entry. The process is removed from its associated ready list by redirecting the appropriate linking threads (pointers). Once removed from the ready list, the process is threaded into the blocked list and its state changed to "blocked" by invocation of the library function LIST_INSERT. Procedure TC_GETWORK is then called to schedule another process for the current virtual processor.

3. TC Advance

The primary purpose of TC_ADVANCE is the broadcasting of some event occurrence. This entails incrementing the eventcount associated with the event, awakening all processes that are waiting for the event, and insuring proper scheduling order by generating any necessary virtual preempt interrupts. The high level design algorithm for TC_ADVANCE is provided in figure 11. The input parameters to TC_ADVANCE are a pointer to a segment's handle and an instance (event number). Initially, TC_ADVANCE locks the APT to prevent the possibility of a race condition. The eventcount identified by the input parameters is then incremented by calling MM_ADVANCE. MM_ADVANCE returns the new value of the eventcount. Once the eventcount has been advanced, TC_ADVANCE awakens all processes awaiting this event occurrence. This is accomplished by checking all processes that are currently in the blocked list. The process' HANDLE and INSTANCE entries are compared with the handle and instance identifying the current event. If they are the same, then the process is awaiting some occurrence of the current event. In such a case, the process' VALUE entry in the APT is compared with the current value of the

```
  TC_ADVANCE   Procedure (HANDLE, INSTANCE)
Begin

  ! Get new eventcount !
  COUNT := MM_ADVANCE (HANDLE, INSTANCE)

  Call  WAIT_LOCK (APT)

  ! Wake up processes !
  PROCESS := BLOCKED_LIST_HEAD

  Do while not end of BLOCKED_LIST
    If (PROCESS.HANDLE = HANDLE) and
       (PROCESS.INSTANCE = INSTANCE) and
       (PROCESS.COUNT <= COUNT)
     then
       Call  LIST_INSERT(READY LIST)
    end if

    PROCESS := PROCESS.NEXT_PROCESS
  end do

  ! Check all ready lists for preempts !
  LOGICAL_CPU_NO := 1

  Do while LOGICAL_CPU_NO <= #NR_CPU
    ! Initialize preempt vector !
    VP_ID := FIRST_VP(LOGICAL_CPU_NO)

    Do for LOOP := 1 to NR_VP(LOGICAL_CPU_NO
      RUNNING_LIST[VP_ID].PREEMPT := #TRUE

      VP_ID := VP_ID + 1
    end do

    ! Find preempt candidates !
    CANDIDATES := 0

    PROCESS := READY_LIST_HEAD(LOGICAL_CPU_NO)
```

Figure 11. TC_ADVANCE Algorithm

```
      VP_ID := FIRST_VP(LOGICAL_CPU_NO)

      Do (for CYCLE = 1 to NR_VP(LOGICAL_CPU_NØ) and
         not end of READY_LIST(LOGICAL_CPU_NO)
        If  PROCESS = #RUNNING
         then
          RUNNING_LIST[VP_ID].PREEMPT := #FALSE
         else
          CANDIDATES := CANDIDATES + 1
        end if

        VP_ID := VP_ID + 1
        PROCESS := PROCESS.NEXT_PROCESS
      end do

      ! Preempt appropriate candidates !
      VP_ID := FIRST_VP(LOGICAL_CPU_NO)

      Do for CHECK := 1 to NR_VP(LOGICAL_CPU_NO)
        If (RUNNING_LIST[VP_ID].PREEMPT = #TRUE) and
           (CANDIDATES > Ø)
         then
          Call  SET_PREEMPT(VP_ID)

          CANDIDATES := CANDIDATES - 1
        end if

        VP_ID := VP_ID + 1
      end do

      LOGICAL_CPU_NO := LOGICAL_CPU_NO + 1
    end do

    Call  UNLOCK(APT)

    Return

  End  TC_ADVANCE
```

Figure 11. TC_ADVANCE Algorithm (Continued)

eventcount. If the process' VALUE is less than or equal to the current eventcount value, the awaited event has occurred and the process is removed from the blocked list and threaded into the appropriate ready list by the library function LIST_INSERT.

Once the blocked list has been checked, it is necessary to reevaluate each ready list to insure that the highest priority processes are running. It is relatively simple to determine if a virtual preempt interrupt is necessary, however, it is considerably more difficult to determine which virtual processor should receive the virtual preempt. To assist in this evaluation, a "count" variable (number of preempts needed) is zeroed and a preempt vector is created on the Kernel stack with an entry for every virtual processor associated with the logical CPU being evaluated. Initially, every entry in the preempt vector is marked "true" indicating that its associated virtual processor is a candidate for preemption. Once the preempt vector is initialized, the first "n" processes on the ready list (where n equals the number of VP's associated with the current logical CPU) are checked for a determination of their state. If a process is found to be "running" then it should not be preempted as processes appear in the ready list in order of priority. When a running process is found, its associated entry in the preempt vector is marked "false." If a process is encountered in the "ready" state then it should be running and the "count" variable is incremented. When the first "n" processes have been checked or when we reach the end of the current ready list (whichever comes first), the entries in the preempt vector are "popped" from the stack. If an entry from the preempt vector is found to be "true", this indicates that its associated virtual processor is a candidate for preemption since it is either bound to a lower priority process, or it is "idle." In such a case, the "count" variable is evaluated to determine if the virtual processor associated with the vector entry should be preempted. If the count exceeds zero, a virtual preempt interrupt is sent to the VP and the count is decremented. Otherwise, no preempt is sent as there is no higher priority process awaiting scheduling.

This preemption algorithm is completed for every ready list in the Active Process Table. Once all ready lists have been evaluated, the APT is unlocked and control is returned to the caller. It is noted that it is not necessary to invoke TC_GETWORK before exiting ADVANCE. If the current VP requires rescheduling, it will have received a virtual preempt interrupt from the preemption algorithm. If this has occurred, the VP will be rescheduled when its running process attempts to leave the Kernel domain and the virtual preempt interrupts are unmasked.

4. Virtual Preempt Handler

VIRTUAL_PREEMPT_HANDLER is the interrupt handler for virtual preempt interrupts. The entry address of VIRTUAL_PREEMPT_HANDLER is maintained in the virtual interrupt vector located in the Inner Traffic Controller. Once invoked, the handler locks the Active Process Table and determines which virtual processor is being preempted by calling RUNNING_VP. The process running on the preempted VP is then set to the "ready" state and TC_GETWORK is invoked to reschedule the virtual processor. When TC_GETWORK returns to VIRTUAL_PREEMPT_HANDLER, the APT is unlocked and a virtual interrupt return is executed. This return is simply a jump to the point in the hardware preempt handler where the virtual interrupts are unmasked. This effects a virtual interrupt return instruction.

5. Remaining Procedures

The remaining two procedures in the Traffic Controller module represent the extended instructions: PROCESS_CLASS and GET_DBR_NUMBER. Both procedures lock the Active Process Table and call RUNNING_VP to determine which virtual processor is executing the current process. The process ID (viz., APT entry Number) is then extracted from the running list. PROCESS_CLASS reads and returns the current process' security access classification from the APT. GET_DBR_NUMBER reads and returns the current process' DBR handle. It should be noted that in general the DBR number provided by procedure GET_DBR_NUMBER is only valid while the APT is locked. Particularly, in the current SASS implementation, the Segment Manager invokes GET_DBR_NUMBER and then passes the obtained DBR number to the Distributed Memory Manager for utilization at that level. In a more general situation, the process associated with the DBR number may have been unloaded before the DBR number was utilized, thus making it invalid. This problem does not arise in SASS as all processes remain loaded for the life of the system.

C. DISTRIBUTED MEMORY MANAGER MODULE

The Distributed Memory Manager module provides an interface between the Segment Manager and the Memory Manager process, manipulates event data in the Global Active Segment Table (G_AST), and dynamically allocates available memory. A detailed description of the Distributed Memory Manager interface to the Memory Manager process was presented by Wells [6]. The remaining extended instruction set is discussed in detail below. The complete PLZ/ASM source listings for the Distributed Memory Manager module is provided in Appendix C.

1. MM Read Eventcount

MM_READ_EVENTCOUNT is invoked by the Event Manager and the Traffic Controller to obtain the current value of the eventcount associated with a particular event. The input parameters to this procedure are a segment handle pointer and an instance (event Number), which together uniquely identify a particular event.

The G_AST is locked and the entry offset of the segment into the G_AST is obtained from the segment's handle. The instance parameter is then validated to determine which eventcount is to be read. If an invalid instance is specified, control is returned to the caller specifying an error condition. Otherwise, the current value of the specified eventcount is read. The G_AST is then unlocked, and the current eventcount value is returned to the caller.

2. MM Advance

MM_ADVANCE is invoked by the Traffic Controller to reflect the occurrence of some event. The input parameters to MM_ADVANCE are a pointer to a segment's handle and a particular instance (event number).

The Global Active Segment Table is locked to prevent a race condition, and the offset of the segment's entry into the G_AST is obtained from the segment handle. The instance parameter is then validated to determine which eventcount is to be advanced. If an invalid instance is specified, an error condition is returned to the caller and no data entries are affected. If the instance value is valid, the appropriate eventcount is incremented, and its new value is returned.

3. MM Ticket

MM_TICKET is invoked by the Event Manager to obtain the current value of the sequencer associated with a specified segment. The input parameter to MM_TICKET is a pointer to a segment's handle.

Initially, MM_TICKET locks the Global Active Segment Table to prevent a race condition. Next the offset of the segment's entry into the G_AST is obtained from the segment handle. The current value of the sequencer for the specified segment is then read and saved as a return parameter to the caller. The sequencer value is then incremented in anticipation of the next ticket request. Once this is complete, the G_AST is unlocked and control is returned to the caller.

4. MM Allocate

The MM_ALLOCATE procedure provided in this implementation is a stub of the MM_ALLOCATE described in the Memory Manager design of Moore and Gary [4].

The primary function of MM_ALLOCATE is the dynamic allocation of fixed size blocks of available memory space. It is invoked in the current implementation by the initialization routines in BOOTSTRAP_LOADER and TC_INIT for the allocation of memory space used in the creation of the Kernel domain and Supervisor domain stack segments and the creation of the Known Segment Tables for user processes. Dynamic reallocation of previously used memory space (viz., garbage collection) is not provided by the MM_ALLOCATE stub in this implementation. All memory allocation required in this implementation is for segments supporting system processes that remain active, and thus allocated, for the entire life of the system. Memory is allocated in blocks of 256 (decimal) bytes of processor local memory (on-board RAM). In this stub allocatable memory is declared at compile time by a data structure (MEM_POOL) that is accessible only by MM_ALLOCATE.

The input parameter to MM_ALLOCATE is the number of blocks of requested memory. This parameter is converted from a block size to the actual number of bytes requested. This computation is made simple since memory is allocated in powers of two. The byte size is obtained by logically shifting left the input parameter *eight times, where eight* is the power of two desired (viz., 256). Once the size of the requested memory is computed, it is necessary to determine the starting address of the memory block(s) to be allocated. To assist in this computation, a variable (NEXT_BLOCK) is used to keep track of the next available block of memory in MEM_POOL. NEXT_BLOCK, which is initialized as zero, provides the offset into the memory being allocated. Once the starting address is obtained, the physical size of the memory allocated is added to NEXT_BLOCK so that the next request for memory allocation will begin at the next free byte of memory in MEM_POOL. This new value of

NEXT_BLOCK is saved and the starting address of the memory for this request is returned to the caller.

D. GATE KEEPER MODULES

The SASS Gate Keeper provides the logical boundary between the Supervisor and the Kernel and isolates the Kernel from the system users, thus making it tamperproof. This is accomplished by means of the hardware system/normal mode and the software ring-crossing mechanism provided by the Gate Keeper. The Gate Keeper is comprised of two separate modules: 1) the USER_GATE module, and 2) the KERNEL_GATE_KEEPER module. These modules are disjoint, with the USER_GATE module residing in the Supervisor domain and the KERNEL_GATE_KEEPER module residing in the Kernel domain. It is important to note that the USER_GATE is a separately linked component in the Supervisor domain and is not linked to the Kernel. The only thing in common between these two modules is a set of constants identifying the valid extended instruction set which the Kernel provides to the users.

The Gate Keeper modules presented in this implenemtation are only stubs as they do not provide all of the functions required of the Gate Keeper. However, the only task not provided in this implementation is the validation of parameters passed from the Supervisor to the Kernel. A detailed description of this parameter validation design is provided by Coleman [3]. In the process management demonstration, the Supervisor stubs are written in PLZ/ASM with all parameters passed by CPU registers. A detailed description of the Gate Keeper modules and the nature of their interfaces is presented below. The PLZ/ASM source listings for the two Gate Keeper modules are provided in Appendix D.

1. User Gate Module

The USER_GATE module provides the interface structure between the user processes in the Supervisor domain and the Kernel. The USER_GATE is comprised of ten procedures (viz., entry points) that correlate on a one to one basis with the ten "user visible" extended instructions (listed in figure 6) provided by the Kernel. The only action performed by each of these procedures is the execution of the "system call" instruction (SC) with a constant value, identifying the particular extended instruction invoked, as the source operand.

The SC instruction is a system trap that forces the hardware into the system mode (Kernel domain) and loads register 15 with the system stack pointer (Kernel domain stack). The current instruction counter value (IC) is pushed onto the Kernel stack along with the current CPU flag control word (FCW). In addition, the system trap instruction is pushed onto the Kernel stack with the upper byte representing the SC instruction and the lower byte representing the SC instruction's source operand (viz., the

Kernel extended instruction code). Togetner, these operations form an interrupt return (IRET) frame as illustrated in figure 9. Once tnis is complete, tne FCW is loaded witn tne FCW value found in tne System Call frame of the Program Status Area (viz., tne hardware "interrupt vector"). Tne structure of tne Program Status Area is illustrated in figure 12. Tne instruction counter is tnen loaded witn the address of the SC instruction trap nandler. This value is also located in tne SC frame of the Program Status Area.

2. Kernel Gate Keeper Module

The system trap nandler for the System Call instruction is the KERNEL_GATE_KEEPER. The address of tne KERNEL_GATE_KEEPER and the Kernel FCW value are placed in the System Call frame of tne Program Status Area by the BOOTSTRAP_LOADER module during initialization. The KERNEL_GATE_KEEPER fetches the extended instruction code from the trap instruction entry in the IRET frame on the Kernel stack. This value is then decoded by a "case" statement to determine wnich extended instruction is to be executed. If the extended instruction code is valid, the appropriate Kernel procedure is invoked. Otnerwise, an error condition is set and no Kernel procedures are not invoked. Once control returns to the KERNEL_GATE_KEEPER, tne CPU registers and normal stack pointer (NSP) value are pusned onto the Kernel stack in preparation for return to the

```
OFFSET

 0   |----------------------|  --|
     |       Reserved       |    |--Frames
 4   |----------------------|  --|
     |    Unimplemented     |
     |     Instruction      |
     |        Trap          |
 8   |----------------------|
     |      Privileged      |
     |     Instruction      |
     |        Trap          |
12   |----------------------|  --|
     |      Kernel FCW      |    |System
     |----------------------|    |--Call
     | Kernel Gate Keeper   |    |Instruction
     |       Address        |    |
16   |----------------------|  --|
     |       Segment        |
     |        Trap          |
20   |----------------------|
     |     Non-Maskable     |
     |      Interrupt       |
24   |----------------------|  --|Hardware
     |      Kernel FCW      |    |Preempt
     |----------------------|    |--
     |PHYS_PREEMPT_HANDLER  |    |(Non-
     |       Address        |    | Vectored
28   |----------------------|  --| Interrupt)
     |     Vectored Int     |
32   |----------------------|
     |          .           |
     |          .           |
     |          .           |
     |----------------------|
```

* NOTE: Offsets represent Program Status Area structure for non-segmented Z8002 microprocessor.

Figure 12. Program Status Area

Supervisor domain. It is noted that this operation would normally occur immediately upon entry into the KERNEL_GATE_KEEPER. In this implementation, however, parameter validation is not accomplished and the CPU registers are used to pass parameters to and from the Kernel only for use by the process management demonstration. In an actual SASS environment, all parameters would be passed in a separate argument list and the CPU registers would appear exactly the same upon leaving the Kernel as they did upon entering the Kernel. This is important to insure that no data or information is leaked from the Kernel by means of the CPU registers.

Control is returned to the Supervisor by means of the return mechanism in the hardware preempt handler. This mechanism is utilized to preclude the necessity of building a separate mechanism for the KERNEL_GATE_KEEPER that would actually perform the very same function. To accomplish this, the KERNEL_GATE_KEEPER executes an unconditional jump to the PREEMPT_RET label in PHYS_PREEMPT_HANDLER. This "jump" to the hardware preempt handler represents a "virtual IRET" instruction providing the same function as the virtual interrupt return described in the discussion of the virtual preempt handler. At this point, the virtual preempt interrupts are unmasked, the normal stack pointer and CPU registers are restored from the stack, and control is returned to the Supervisor by execution of the IRET instruction.

E. SUMMARY

The implementation of process management functions for the SASS has been presented in this chapter. The implementation was discussed in terms of the Event Manager, Traffic Controller, Distributed Memory Manager, and Gate Keeper modules.

Chapter V will present the conclusions drawn from this work and suggestions for future work derived from this thesis.

V. CONCLUSION

The implementation of process management for the security Kernel of a secure archival storage system has been presented. The process management functions presented provide a logical and efficient means of process creation, control, and scheduling. In addition, a simple but effective mechanism for inter-process communication, based on the eventcount and sequencer primitives, was created. Work was also completed in the area of Kernel database initialization and a Gate Keeper stub to allow for dual domain operation.

The design for this implementation was based on the Zilog Z8001 sixteen bit segmented microprocessor [17] used in conjunction with the Zilog Z8010 Memory Management Unit [18]. The actual implementation of process management for the SASS was conducted on the Advanced Micro Computers Am96/4116 MonoBoard Computer [19] featuring the AmZ8002 sixteen bit non-segmented microprocessor. Segmentation hardware was simulated by a software Memory Management Unit Image.

This implementation was effected specifically to support the Secure Archival Storage System (SASS) [21]. However, the implementation is based on a family of Operating Systems [1] designed with a primary goal of providing multilevel information security. The loop free modular design utilized in this implementation easily facilitates any required expansion or modification for other family members. In addition, this implementation fully supports a multiprocessor design. While the process management implementation appears to perform correctly, it has not been subjected to a formal test plan. Such a test plan should be developed and implemented before kernel verification is begun.

A. FOLLOW ON WORK

There are several possible areas in the SASS design that would be immediately suitable for continued research. In the area of hardware, this includes, the establishment of a multiprocessor environment, hardware initialization, and interfacing to the host computers and secondary storage. Further work in the Kernel includes the actual implementation of the memory manager process, and the refinement of the Gate Keeper and Kernel intialization structures. The implementation of the Supervisor has not been addressed to date. Its areas of research include the implementation of the File Manager and Input/Output processes, and the final design and implementation of the SASS-Hosts protocols.

Other areas that could also prove interesting in relation to the SASS include the implementation of dynamic memory management, the support of multilevel hosts, dynamic process creation and deletion, and the provision of constructive work to be performed by the Idle process.

APPENDIX A - EVENT MANAGER LISTINGS

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

        $LISTON $TTY

        EVENT_MGR       MODULE

        CONSTANT
          TRUE                      := 1
          FALSE                     := 0
          READ_ACCESS               := 1
          WRITE_ACCESS              := 0
          SUCCEEDED                 := 2
          SEGMENT_NOT_KNOWN         := 28
          ACCESS_CLASS_NOT_EQ       := 33
          ACCESS_CLASS_NOT_GE       := 41
          KST_SEG_NO                := 2
          NR_OF_KSEGS               := 10
          MAX_NO_KST_ENTRIES        := 54
          NOT_KNOWN                 := %FF

        TYPE
          H_ARRAY       ARRAY[3 WORD]

          KST_REC    RECORD
           [MM_HANDLE   H_ARRAY
            SIZE        WORD
            ACCESS_MODE BYTE
            IN_CORE     BYTE
            CLASS       LONG
            M_SEG_NO    SHORT_INTEGER
            ENTRY_NUMBER SHORT_INTEGER
           ]

        EXTERNAL
          MM_TICKET                 PROCEDURE
          MM_READ_EVENTCOUNT        PROCEDURE
          TC_ADVANCE                PROCEDURE
          TC_AWAIT                  PROCEDURE
          PROCESS_CLASS PROCEDURE
          CLASS_EQ                  PROCEDURE
          CLASS_GE                  PROCEDURE
          ITC_GET_SEG_PTR           PROCEDURE
          INTERNAL

          $SECTION EM_KST_DCL
          ! NOTE: THIS SECTION IS AN "OVERLAY"
            OR "FRAME" USED TO DEFINE THE
            FORMAT OF THE KST.  NO STORAGE IS
            ASSIGNED BUT RATHER THE KST IS
            STORED IN A SEPARATELY OBTAINED
            AREA. (A SEGMENT SET ASIDE FOR IT)!

            $ABS 0
0000        KST   ARRAY[MAX_NO_KST_ENTRIES KST_REC]
```

```
            GLOBAL
            $SECTION EM_GLB_PROC

0000             READ                      PROCEDURE
                !******************************
                 * READS SPECIFIED EVENTCOUNT *
                 * AND RETURNS IT'S VALUE TO  *
                 * THE CALLER                 *
                 ******************************
                 * PARAMETERS:                *
                 *  R1: SEGMENT #             *
                 *  R2: INSTANCE              *
                 ******************************
                 * RETURNS:                   *
                 *  R0: SUCCESS CODE          *
                 *  R4: EVENTCOUNT            *
                 ******************************!

                 ENTRY
                  ! SAVE INSTANCE !
0000 93F2         PUSH   @R15, R2

                  ! "READ" ACCESS REQUIRED !
0002 2102         LD      R2, #READ_ACCESS
0004 0001
                  ! GET SEG HANDLE & VERIFY ACCESS !
0006 5F00         CALL    CONVERT_AND_VERIFY !R1:SEG #
0008 0000'
                                              R2:REQ. ACCESS
                                              RETURNS:
                                              R0:SUCCESS CODE
                                              R1:HANDLE PTR!
000A 0B00         CP      R0, #SUCCEEDED
000C 0002
                  IF EQ !ACCESS PERMITTED!
000E 5E0E          THEN  !READ EVENTCOUNT!
0010 001C'
                    !RESTORE INSTANCE!
0012 97F2           POP   R2, @R15
0014 5F00           CALL MM_READ_EVENTCOUNT !R1:HPTR
0016 0000*
                                              R2:INSTANCE
                                              RETURNS:
                                              R0:SUCCESS CODE
                                              RR4:EVENTCOUNT!
0018 5E08          ELSE  !RESTORE SP!
001A 001E'
001C 97F2           POP   R2, @R15
                  FI
001E 9E08         RET
0020             END READ
```

```
0020         TICKET                          PROCEDURE
            !********************************
             * RETURNS CURRENT VALUE OF     *
             * TICKET TO CALLER AND INCRE-  *
             * MENTS SEQUENCER FOR NEXT     *
             * TICKET OPERATION             *
             ********************************
             * PARAMETERS:                  *
             *  R1: SEGMENT #               *
             ********************************
             * RETURNS:                     *
             *  R0: SUCCESS CODE            *
             *  RR4: TICKET VALUE           *
             ********************************!

             ENTRY
              ! GET SEG HANDLE & VERIFY ACCESS !
              ! "WRITE" ACCESS REQUIRED !
0020 2102     LD     R2, #WRITE_ACCESS
0022 0000
0024 5F00     CALL   CONVERT_AND_VERIFY !R1:SEG #
0026 0000'
                                         R2:ACCESS REQ
                                         RETURNS:
                                         R0:SUCCESS CODE
                                         R1:HANDLE PTR!
0028 0B00     CP     R0, #SUCCEEDED
002A 0002
              IF EQ !ACCESS PERMITTED!
002C 5E0E      THEN  ! GET TICKET !
002E 0038'
0030 5F00       CALL  MM_TICKET !R1:HANDLE PTR
0032 0000*
                                    RETURNS:
                                    RR4:TICKET!
                ! RSTORE SUCCESS CODE !
0034 2100       LD    R0, #SUCCEEDED
0036 0002
              FI
0038 9E08     RET
003A         END TICKET
```

```
003A         AWAIT                    PROCEDURE
            !*******************************
             * CURRENT EVENTCOUNT VALUE IS *
             * COMPARED TO USER SPECIFIED  *
             * VALUE. IF USER VALUE IS     *
             * GREATER THAN CURRENT EVENT- *
             * COUNT VALUE THEN PROCESS IS *
             * "BLOCKED" UNTIL THE DESIRED *
             * EVENT OCCURS.               *
             *******************************
             * PARAMETERS:                 *
             *  R1: SEGMENT #              *
             *  R2: INSTANCE (EVENT #)     *
             *  RR4: SPECIFIED VALUE       *
             *******************************
             * RETURNS:                    *
             *  R0: SUCCESS CODE           *
             *******************************!

             ENTRY
              ! SAVE DESIRED EVENTCOUNT VALUE !
003A 91F4     PUSHL  @R15, RR4
              ! SAVE INSTANCE !
003C 93F2     PUSH   @R15, R2
              ! "READ" ACCESS REQUIRED !
003E 2102     LD     R2, #READ_ACCESS
0040 0001
              ! GET SEG HANDLE & VERIFY ACCESS !
0042 5F00     CALL   CONVERT_AND_VERIFY !R1:SEG #
0044 0000'
                                         R2:ACCESS REQ
                                         RETURNS:
                                         R0:SUCCESS CODE
                                         R1:HANDLE PTR!
0046 0B00     CP     R0, #SUCCEEDED
0048 0002
              IF EQ ! ACCESS PERMITTED !
004A 5E0E      THEN ! AWAIT EVENT OCCURRENCE !
004C 005A'
                ! RESTORE INSTANCE !
004E 97F2       POP  R2, @R15
                ! RESTORE SPECIFIED VALUE !
0050 95F4       POPL RR4, @R15
0052 5F00       CALL TC_AWAIT !R1:HANDLE PTR
0054 0000*
                              R2:INSTANCE
                              RR4:VALUE
                              RETURNS:
                              R0:SUCCESS CODE!
```

```
0056 5E08        ELSE !RESTORE STACK!
0058 005E
005A 95F4          POPL  RR4, @R15
005C 97F2          POP   R2, @R15
                 FI
005E 9E08        RET
0060            END AWAIT
```

```
0060          ADVANCE                    PROCEDURE
             !********************************
              * SIGNALS THE OCCURRENCE OF    *
              * SOME EVENT.  EVENTCOUNT IS   *
              * INCREMENTED AND THE TRAFFIC  *
              * CONTROLLER IS INVOKED TO     *
              * AWAKEN ANY PROCESS AWAITING  *
              * THE OCCURRENCE.              *
              ********************************
              * PARAMETERS:                  *
              *  R1: SEGMENT #               *
              *  R2: INSTANCE (EVENT #)      *
              ********************************
              * RETURNS:                     *
              *  RØ: SUCCESS CODE            *
              ********************************!

              ENTRY
               ! SAVE INSTANCE !
0060 93F2      PUSH  @R15, R2

               ! GET SEG HANDLE & VERIFY ACCESS !
               ! "WRITE" ACCESS REQUIRED !
0062 2102      LD    R2, #WRITE_ACCESS
0064 0000
0066 5F00      CALL  CONVERT_AND_VERIFY !R1:SEG #
0068 0000'
                                         R2:ACCESS REC
                                         RETURNS:
                                         RØ:SUCCESS CODE
                                         R1:HANDLE PTR!
006A 0B00      CP    RØ, #SUCCEEDED
006C 0002
               IF EQ ! ACCESS PERMITTED !
006E 5E0E       THEN ! ADVANCED EVENTCOUNT !
0070 007C'
                 ! RESTORE INSTANCE !
0072 97F2        POP   R2, @R15

0074 5F00        CALL  TC_ADVANCE !R1:HANDLE PTR
0076 0000*
                                    R2:INSTANCE
                                    RETURNS:
                                    RØ:SUCCESS CODE!
0078 5E08       ELSE !RESTORE STACK!
007A 007E'
007C 97F2        POP   R2, @R15
               FI
007E 9E08      RET
0080          END ADVANCE
```

```
                 INTERNAL
                 $SECTION EM_INT_PROC

0000              CONVERT_AND_VERIFY              PROCEDURE
                  !***************************************
                  * CONVERTS SEGMENT NUMBER TO KST INDEX*
                  * AND EXTRACTS SEGMENT'S HANDLE FROM  *
                  * KST. IF SUCCESSFUL, THEN ACCESS     *
                  * CLASS OF SUBJECT IS CHECKED AGAINST *
                  * ACCESS CLASS OF OBJECT TO INSURE    *
                  * THAT ACCESS IS PERMITTED.           *
                  ***************************************
                  * PARAMETERS:                         *
                  *  R1: SEGMENT NUMBER                 *
                  *  R2: ACCESS REQUESTED               *
                  ***************************************
                  * RETURNS:                            *
                  *  R0: SUCCESS CODE                   *
                  *  R1: HANDLE POINTER                 *
                  ***************************************!

                  ENTRY
                   ! SAVE REQUESTED ACCESS !
0000 93F2          PUSH   @R15, R2
                   ! GET SEGMENT HANDLE !
0002 5F00          CALL   GET_HANDLE !R1:SEG #
0004 0062'
                                       RETURNS:
                                       R0:SUCCESS CODE
                                       R4:HANDLE PTR
                                       R5:CLASS PTR!
0006 0B00          CP     R0, #SUCCEEDED
0008 0002
                   IF EQ   ! SEGMENT IS KNOWN !
000A 5E0E           THEN   ! VERIFY ACCESS !
000C 005E'
                     ! SAVE HANDLE & CLASS PTR !
000E 91F4            PUSHL @R15, RR4
                     ! GET SUBJECT'S SAC !
0010 5F00            CALL  PROCESS_CLASS !RETURNS:
0012 0000*
                                             RR2:PROC CLASS!
                     ! RETRIEVE SEG CLASS POINTER !
0014 95F0            POPL   RR0, @R15
                     ! GET SEGMENT'S CLASS !
0016 1414            LDL    RR4, @R1
                     ! RETRIEVE REQUESTED ACCESS !
0018 97F1            POP    R1, @R15
                     ! SAVE HANDLE POINTER !
001A 93F0            PUSH  @R15, R0
                     ! CHECK ACCESS CLEARANCE !
```

```
001C 0B01       CP    R1, #WRITE_ACCESS
001E 0000
                IF EQ ! WRITE ACCESS REQUESTED !
0020 5E0E        THEN
0022 0040'
0024 5F00          CALL  CLASS_EQ !RR2:PROCESS CLASS
0026 0000*
                                 RR4:SEGMENT CLASS
                                 RETURNS:
                                 R1: CONDITION CODE!
0028 0B01          CP    R1, #FALSE
002A 0000
                   IF EQ !ACCESS NOT PERMITTED!
002C 5E0E           THEN
002E 0038'
0030 2100            LD  R0, #ACCESS_CLASS_NOT_EQ
0032 0021
0034 5E08           ELSE !ACCESS PERMITTED!
0036 003C'
0038 2100            LD  R0, #SUCCEEDED
003A 0002
                   FI
003C 5E08        ELSE ! READ ACCESS REQUESTED !
003E 0058'
0040 5F00          CALL  CLASS_GE !RR2:PROCESS CLASS
0042 0000*
                                 RR4:SEGMENT CLASS
                                 RETURNS:
                                 R1:CONDITION CODE!
0044 0B01          CP    R1, #FALSE
0046 0000
                   IF EQ !ACCESS NOT PERMITTED!
0048 5E0E           THEN
004A 0054'
004C 2100            LD  R0, #ACCESS_CLASS_NOT_GE
004E 0029
0050 5E08           ELSE !ACCESS PERMITTED!
0052 0058'
0054 2100            LD  R0, #SUCCEEDED
0056 0002
                   FI
                FI
                ! RETRIEVE HANDLE POINTER !
0058 97F1       POP   R1, @R15
005A 5E08     ELSE
005C 0060'
                ! RESTORE STACK !
005E 97F2       POP   R2, @R15
              FI
0060 9E08     RET
0062         END CONVERT_AND_VERIFY
```

```
0062            GET HANDLE                  PROCEDURE
               !**********************************
                * CONVERTS SEGMENT NUMBER TO     *
                * KST INDEX AND DETERMINES IF    *
                * SEGMENT IS KNOWN.  IF KNOWN    *
                * POINTER TO SEGMENT HANDLE      *
                * AND POINTER TO SEGMENT CLASS*
                * ARE RETURNED.                  *
                **********************************
                * PARAMETERS:                    *
                *  R1: SEGMENT NUMBER            *
                **********************************
                * RETURNS:                       *
                *  R0: SUCCESS CODE              *
                *  R4: HANDLE POINTER            *
                *  R5: CLASS POINTER             *
                **********************************!

                ENTRY
                 ! CONVERT SEGMENT # TO KST INDEX # !
0062 0301        SUB   R1, #NR_OF_KSEGS
0064 000A
                 ! VERIFY KST INDEX !
0066 2100        LD    R0, #SUCCEEDED
0068 0002
006A 0B01        CP    R1, #0
006C 0000
                 IF LE !INDEX NEGATIVE!
006E 5E0A         THEN
0070 007A'
0072 2100          LD    R0, #SEGMENT_NOT_KNOWN
0074 001C
0076 5E08         ELSE !INDEX POSITIVE!
0078 0086'
007A 0B01          CP    R1, #MAX_NO_KST_ENTRIES
007C 0036
                   IF  GT !EXCEEDS MAXIMUM INDEX!
007E 5E02           THEN  !INVALID INDEX!
0080 0086'
0082 2100            LD    R0, #SEGMENT_NOT_KNOWN
0084 001C
                   FI
                 FI
0086 0B00        CP    R0, #SUCCEEDED
0088 0002
                 IF EQ  !INDEX VALID!
008A 5E0E         THEN
008C 00BE'
                   ! SAVE KST INDEX !
008E 93F1          PUSH  @R15, R1
                   ! GET KST ADDRESS !
```

```
0090 2101       LD    R1, #KST_SEG_NO
0092 0002
0094 5F00       CALL  ITC_GET_SEG_PTR !R1:KST_SEG_NO
0096 0000*
                                       RETURNS:
                                       R0:KST ADDR!
                ! RETRIEVE KST INDEX # !
0098 97F3       POP   R3, @R15

                ! CONVERT KST INDEX # TO KST OFFSET !
009A 1902       MULT  RR2, #SIZEOF KST_REC
009C 0010
                ! COMPUTE KST ENTRY ADDRESS !
009E 8103       ADD   R3, R0
                ! SEE IF SEGMENT IS KNOWN !
00A0 4D31       CP    KST.M_SEG_NO(R3), #NOT_KNOWN
00A2 000E
00A4 00FF
                IF EQ !SEGMENT NOT KNOWN!
00A6 5E0E        THEN
00A8 00B2'
00AA 2100         LD  R0, #SEGMENT_NOT_KNOWN
00AC 001C
00AE 5E08        ELSE !SEGMENT KNOWN!
00B0 00BE'
00B2 2100         LD  R0, #SUCCEEDED
00B4 0002
                  ! GET HANDLE POINTER !
00B6 7634         LDA   R4, KST.MM_HANDLE(R3)
00B8 0000
                  ! GET CLASS POINTER !
00BA 7635         LDA   R5, KST.CLASS(R3)
00BC 000A
                FI
              FI
00BE 9E08     RET
00C0         END GET_HANDLE
            END EVENT_MGR
```

APPENDIX B - TRAFFIC CONTROLLER LISTINGS

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

          $LISTON $TTY
          TC MODULE


           CONSTANT

           ! ******** SYSTEM PARAMETERS ******** !
                  NR_PROC            := 4
                  VP_NR              := 2
                  NR_CPU             := 2
                  NR_KST             := 54

            ! ******** SYSTEM CONSTANTS ******** !
                 RUNNING        := 0
                 READY          := 1
                 BLOCKED        := 2
                 IDLE_PROC      := %DDDD
                 NIL            := %FFFF
                 INVALID        := %EEEE
                 KERNEL_STACK   := 1
                 USER_STACK     := 3
                 KST_SEG        := 2
                 KST_LIMIT      := 1
                 USER_FCW       := %1800
                 WRITE          := 0
                 !INDICATES LOWEST SYSTEM
                  SECURITY CLASS!
                 SYSTEM_LOW     := 0
                 STK_OFFSET     := %FF
                 REMOVED        := %ABCD
                 TRUE           := 1
                 FALSE          := 0
                 SUCCEEDED      := 2

          TYPE
           AP_PTR       WORD
           VP_PTR       WORD
           ADDRESS      WORD
           H_ARRAY      ARRAY[3 WORD]
```

```
AP_TABLE RECORD
      [NEXT_AP          AP_PTR
       DBR              WORD
       SAC              LONG
       PRI              INTEGER
       STATE            INTEGER
       AFFINITY         WORD
       VP_ID            VP_PTR
       HANDLE           H_ARRAY
       INSTANCE         WORD
       VALUE            LONG
       FILL_2           ARRAY[2 WORD]
      ]

RUN_ARRAY       ARRAY[VP_NR  AP_PTR]
RDY_ARRAY       ARRAY[NR_CPU AP_PTR]
AP_DATA         ARRAY[NR_PROC AP_TABLE]
VP_DATA         RECORD
  [NR_VP        ARRAY[NR_CPU WORD]
   FIRST        ARRAY[NR_CPU VP_PTR]
  ]

KST_REC         RECORD
 [MM_HANDLE     H_ARRAY
  SIZE          WORD
  ACCESS        BYTE
  IN_CORE       BYTE
  CLASS         LONG
  M_SEG_NO      SHORT_INTEGER
  ENTRY_NUM     SHORT_INTEGER
 ]

EXTERNAL
   K_LOCK                  PROCEDURE
   K_UNLOCK                PROCEDURE
   SET_PREEMPT             PROCEDURE
   SWAP_VDBR               PROCEDURE
   IDLE                    PROCEDURE
   RUNNING_VP              PROCEDURE
   CREATE_INT_VEC          PROCEDURE
   LIST_INSERT             PROCEDURE
   ALLOCATE_MMU            PROCEDURE
   MM_ALLOCATE             PROCEDURE
   UPDATE_MMU_IMAGE        PROCEDURE
   CREATE_STACK            PROCEDURE
   MM_ADVANCE              PROCEDURE
   MM_READ_EVENTCOUNT      PROCEDURE
   G_AST_LOCK              WORD
   PREEMPT_RET             LABEL
```

```
          $SECTION TC_DATA
          INTERNAL
0000       APT        RECORD
             [ LOCK               WORD
               RUNNING_LIST       RUN_ARRAY
               READY_LIST         RDY_ARRAY
               BLOCKED_LIST       AP_PTR
               FILL_3             LONG
               VP                 VP_DATA
               FILL               ARRAY[4 WORD]
               AP                 AP_DATA
             ]

          !THESE VARIABLES ARE USED DURING TC
           INITIALIZATION TO SPECIFY AVAILABLE
           ENTRIES IN THE APT, AND ARE INITIAI-
           IZED BY TC_INIT IN THIS IMPLEMENTATION!
00A0      NEXT_VP    WORD
00A2      APT_ENTRY  WORD

         $SECTION  TC_LOCAL
         $ABS 0
         !NOTE: USED AS OVERLAY ONLY!
0000      ARG_LIST      RECORD
            [REG          ARRAY[13 WORD]
             IC           WORD
             CPU_ID       WORD
             SAC1         LONG
             PRI1         WORD
             USR_STK      WORD
             KER_STK      WORD
             KST1         LONG
            ]

         $ABS 0
         !NOTE: USED AS STACK FRAME FOR
          STORAGE OF TEMPORARY VARIABLES
          FOR CREATE_PROCESS.!
0000      CREATE     RECORD
            [ARG_PTR   WORD
             DBR_NUM   WORD
             LIMITS    WORD
             SEG_ADDR  ADDRESS
             N_S_P     WORD
            ]

         $ABS 0
0000      HANDLE_VAL    RECORD
              [HIGH   LONG
               LOW   WORD
              ]
          !THE FOLLOWING DECLARATION IS UTILIZED
           AS A STACK FRAME FOR STORAGE OF
           TEMPORARY VARIABLES UTILIZED BY
           TC_ADVANCE AND TC_AWAIT.!
          $ABS 0
0000       TEMP      RECORD
             [HANDLE_PTR     WORD
              EVENT_NR       WORD
              EVENT_VAL      LONG
              ID_VP          WORD
              CPU_NUM        WORD
              HANDLE_HIGH    LONG
              HANDLE_LOW     WORD
             ]

          $SECTION TC_KST_DCL
           !NOTE: KST DECLARATION IS USED HERE
            TO SUPPORT KST INITIALIZATION FOR
            THIS DEMONSTRATION ONLY.  THIS
            DECLARATION AND INITIALIZATION
            SHOULD EXIST AT THE SEGMENT MANAGER
            LEVEL AND THUS SHOULD BE REMOVED
            UPON IMPLEMENTATION OF SYSTEM
            INITIALIZATION.!
              $ABS 0
0000           KST  ARRAY[NR_KST KST_REC]
```

```
             $SECTION TC_INT_PROC
0000           TC_GETWORK           PROCEDURE
              !****************************
               * PROVIDES GENERAL MANAGE-  *
               * MENT OF USER PROCESSES BY *
               * EFFECTING PROCESS SCHEDU- *
               * LING ON VIRTUAL PROCESSORS*
               ****************************
               * PARAMETERS:               *
               *  R1: CURRENT VP ID        *
               *  R3: LOGICAL CPU #        *
               ****************************
               * LOCAL VARIABLES:          *
               *  R2: NEXT READY PROCESS   *
               *  R4: AP PTR               *
               ****************************!

               ENTRY
                ! FIND FIRST READY PROCESS !
0000 6132       LD    R2, APT.READY_LIST(R3)
0002 0006'
                GET_READY_AP:
                DO  !WHILE NOT (END OF LIST OR READY)!
0004 0B02        CP    R2, #NIL
0006 FFFF
0008 5E0E        IF EQ !NO READY PROCESS! THEN
000A 0010'
000C 5E08         EXIT FROM GET_READY_AP
000E 0026'
                 FI
0010 4D21        CP    APT.AP.STATE(R2), #READY
0012 002A'
0014 0001
0016 5E0E        IF EQ !PROCESS READY! THEN
0018 001E'
001A 5E08         EXIT FROM GET_READY_AP
001C 0026'
                 FI
                 ! GET NEXT AP FROM LIST !
001E 6124        LD    R4, APT.AP.NEXT_AP(R2)
0020 0020'
0022 A142        LD    R2, R4
0024 E8EF       OD
0026 0B02       CP    R2,#NIL
0028 FFFF
002A 5E0E       IF EQ ! IF NO PROCESSES READY ! THEN
002C 003C'
                 ! LOAD IDLE PROCESS !
002E 4D15        LD    APT.RUNNING_LIST(R1), #IDLE_PROC
0030 0002'
0032 DDDD
```

```
0034 5F00      CALL IDLE
0036 0000*
0038 5E08     ELSE
003A 0052'
               ! LOAD FIRST READY AP !
003C 6F12      LD   APT.RUNNING_LIST(R1), R2
003E 0002'
0040 4D25      LD   APT.AP.STATE(R2), #RUNNING
0042 002A'
0044 0000
0046 6F21      LD   APT.AP.VP_ID(R2), R1
0048 002E'
004A 6121      LD   R1, APT.AP.DBR(R2)
004C 0022'
004E 5F00      CALL SWAP_VDBR  !(R1:DBR)!
0050 0000*
              FI
0052 9E08     RET
0054        END TC_GETWORK
```

```
0054        VIRTUAL_PREEMPT_HANDLER    PROCEDURE
              !****************************
               * LOADS FIRST READY AP     *
               * IN RESPONSE TO PREEMPT   *
               * INTERRUPT                *
               ****************************!

               ENTRY
                !** CALL WAIT_LOCK (APT^.LOCK) **!
                !** RETURNS WHEN PROCESS HAS LOCKED APT **!
0054 7604       LDA   R4, APT.LOCK
0056 0000'
0058 5F00       CALL  K_LOCK
005A 0000*
                ! GET RUNNING_VP ID !
005C 5F00       CALL  RUNNING_VP  !RETURNS:
005E 0000*
                                     R1:VP_ID
                                     R3:CPU #!

                ! GET AP !
0060 6112       LD      R2, APT.RUNNING_LIST(R1)
0062 0002'

                ! IF NOT AN IDLE PROCESS, SET IT TO READY !
0064 0B02       CP      R2, #IDLE_PROC
0066 DDDD
0068 5E06       IF NE ! NOT IDLE !  THEN
006A 0072'
006C 4D25        LD    APT.AP.STATE(R2), #READY
006E 002A'
0070 0001
                FI

                ! LOAD FIRST READY PROCESS !
0072 5F00       CALL TC_GETWORK  !R1:VP_ID
0074 0000'
                                    R3:CPU #!

               !NOTE: THIS IS THE INITIAL POINT OF
                EXECUTION FOR USER PROCESSES.!
               VIRT_PREEMPT_RETURN:
                !** CALL UNLOCK (APT^.LOCK) **!
                !** RETURNS WHEN PROCESS HAS UNLOCKED APT **!
                !** AND ADVANCED ON THIS EVENT **!
0076 7604       LDA   R4, APT.LOCK
0078 0000'
007A 5F00       CALL  K_UNLOCK
007C 0000*
```

```
                  ! PERFORM A VIRTUAL INTERRUPT RETURN !
                  !NOTE: THIS JUMP EFFECTS A VIRTUAL
                   IRET INSTRUCTION.!
007E 5E08         JP   PREEMPT_RET
0080 0000*
0082             END VIRTUAL_PREEMPT_HANDLER
```

```
                  GLOBAL
                  $SECTION TC_GLB_PROC
0000                 TC_INIT                 PROCEDURE
                    !*****************************
                     * INITIALIZES APT HEADER    *
                     * AND VIRTUAL INT VECTOR    *
                     *****************************
                     * PARAMETERS:               *
                     *  R1: CPU_ID               *
                     *  R2: NR_VP                *
                     *****************************!

                     ENTRY
                      ! NOTE: THE NEXT FOUR VALUES ARE
                        ONLY TO BE INITIALIZED ONCE. !
0000 4D05             LD   NEXT_VP, #0
0002 00A0'
0004 0000
0006 4D05             LD   APT_ENTRY, #0
0008 00A2'
000A 0000
000C 4D05             LD   APT.BLOCKED_LIST, #NIL
000E 000A'
0010 FFFF
0012 4D08             CLR  APT.LOCK
0014 0000'
                      !****************************************
                       NOTE: THE FOLLOWING CODE IS INCLUDED
                       ONLY FOR SIMULATION OF A MULTIPROCESSOR
                       ENVIRONMENT.  THIS IS TO INSURE THAT THE
                       READY LIST(S) AND VP DATA OF THE SIMULATED
                       CPU(S) ARE PROPERLY INITIALIZED.  IN AN
                       ACTUAL MULTIPROCESSOR ENVIRONMENT, THIS
                       BLOCK OF CODE SHOULD BE REMOVED.
                       ****************************************!
0016 2104                   LD    R4, #0
0018 0000
                            DO
001A 0B04                    CP    R4, #NR_CPU*2
001C 0004
                             IF EQ !ALL LISTS INITIALIZED!
001E 5E0E                     THEN EXIT
0020 0026'
0022 5E08
0024 0036'
                             FI
                       ! INITIALIZE READY LISTS AS EMPTY !
0026 4D45              LD     APT.READY_LIST(R4), #NIL
0028 0006'
002A FFFF
                       ! INITIALLY MARK ALL LOGICAL CPU'S
                         AS HAVING 1 VP.  THIS IS NECESSARY
                         TO INSURE TC_ADVANCE WILL FUNCTION
                         PROPERLY, AS IT EXPECTS EVERY CPU
                         TO HAVE AT LEAST 1 VP. !
002C 4D45              LD     APT.VP.NR_VP(R4), #1
002E 0010'
0030 0001
0032 A941              INC    R4, #2
0034 E8F2             OD
                  ! END MULTIPROCESSOR SIMULATION CODE.
                  ****************************************!
0036 6F12         LD   APT.VP.NR_VP(R1), R2
0038 0010'
003A 6103         LD   R3, NEXT_VP
003C 00A0'
003E 6F13         LD   APT.VP.FIRST(R1), R3
0040 0014'
                 ! RECOMPUTE NEXT_VP VALUE FOR TC
                  INITIALIZATION OF NEXT LOGICAL
                  CPU. !
0042 A125         LD   R5, R2
0044 1904         MULT RR4, #2
0046 0002
0048 8153         ADD  R3, R5
004A 6F03         LD   NEXT_VP, R3
004C 00A0'
                  ! INITIALIZE RUNNING LIST !
004E 6113         LD   R3, APT.VP.FIRST(R1)
0050 0014'
                  DO
0052 0B02          CP   R2, #0
0054 0000
0056 5E0E          IF EQ THEN EXIT FI
0058 005E'
005A 5E08
005C 006A'
005E 4D35          LD   APT.RUNNING_LIST(R3), #IDLE_PROC
0060 0002'
0062 DDDD
0064 A931          INC  R3, #2
0066 AB20          DEC  R2, #1
0068 E8F4         OD
006A 4D15         LD   APT.READY_LIST(R1), #NIL
006C 0006'
006E FFFF
```

```
0070 2101     LD    R1, #0
0072 0000
              ! ENTRY ADDRESS !
0074 7602     LDA   R2, VIRTUAL_PREEMPT_HANDLER
0076 0054'
0078 5F00     CALL    CREATE_INT_VEC
007A 0000*
                      !R1:VIRTUAL INTERRUPT #
                       R2:INTERRUPT HANDLER ADDRESS!
007C 9E08     RET
007E         END TC_INIT
```

```
007E        CREATE PROCESS  PROCEDURE
           !**************************
            * CREATES USER PROCESS   *
            * DATABASES AND APT      *
            * ENTRIES                *
            **************************
            * PARAMETERS:            *
            *  R14: ARGUMENT PTR     *
            **************************!

            ENTRY
             !NOTE: THIS PROCEDURE IS A STUB TO ALLOW
              PROCESS INITIALIZATION FOR THIS
              DEMONSTRATION.!
             ! ESTABLISH STACK FRAME FOR LOCAL
               VARIABLES. !
007E 030F    SUB    R15, #SIZEOF CREATE
0080 000A
             ! STORE INPUT ARGUMENT POINTER !
0082 6FFE    LD     CREATE.ARG_PTR(R15), R14
0084 0000
             ! LOCK APT !
0086 7604    LDA    R4, APT.LOCK
0088 0000'
008A 5F00    CALL   K_LOCK
008C 0000*
             ! RETURNS WHEN APT IS LOCKED !
             ! CREATE MMU ENTRY FOR PROCESS !
008E 5F00    CALL   ALLOCATE_MMU !RETURNS:
0090 0000*
                                     R0: DBR #!
             ! GET NEXT AVAILABLE ENTRY IN APT !
0092 6101    LD     R1, APT_ENTRY
0094 00A2'
             ! COMPUTE APT OFFSET !
0096 2102    LD     R2, #SIZEOF AP_TABLE
0098 0020
009A 8112    ADD    R2, R1
             ! SAVE NEXT AVAILABLE APT ENTRY !
009C 6F02    LD     APT_ENTRY, R2
009E 00A2'
             ! CREATE APT ENTRY FOR PROCESS !
00A0 4D15    LD     APT.AP.NEXT_AP(R1), #NIL
00A2 0020'
00A4 FFFF
00A6 6F10    LD     APT.AP.DBR(R1), R0
00A8 0022'
             ! GET PROCESS CLASS !
00AA 54E2    LDL    RR2, ARG_LIST.SAC1(R14)
00AC 001E
00AE 5D12    LDL    APT.AP.SAC(R1), RR2
```

```
00B0 0024'
               ! GET PROCESS PRIORITY !
00B2 61E2      LD    R2, ARG_LIST.PRI1(R14)
00B4 0022
00B6 6F12      LD    APT.AP.PRI(R1), R2
00B8 0028'
               ! GET LOGICAL CPU # !
00BA 61E2      LD    R2, ARG_LIST.CPU_ID(R14)
00BC 001C
00BE 6F12      LD    APT.AP.AFFINITY(R1), R2
00C0 002C'
               !THREAD IN LIST AND MAKE READY!
00C2 7623      LDA   R3, APT.READY_LIST(R2)
00C4 0006'
00C6 7604      LDA   R4, APT.AP.NEXT_AP
00C8 0020'
00CA 7605      LDA   R5, APT.AP.PRI
00CC 0028'
00CE 7606      LDA   R6, APT.AP.STATE
00D0 002A'
00D2 2107      LD    R7, #READY
00D4 0001
00D6 AD21      EX    R1, R2
               ! SAVE DBR # !
00D8 6FF0      LD    CREATE.DBR_NUM(R15), R0
00DA 0002
00DC 5F00      CALL  LIST_INSERT
00DE 0000*
                     !R2: OBJ ID
                      R3: LIST HEAD PTR
                      R4: NEXT OBJ PTR
                      R5: PRIORITY PTR
                      R6: STATE PTR
                      R7: STATE!
               ! UNLOCK APT !
00E0 7604      LDA   R4, APT.LOCK
00E2 0000'
00E4 5F00      CALL  K_UNLOCK
00E6 0000*
               !CREATE USER STACK!
               ! RESTORE ARGUMENT POINTER !
00E8 61FE      LD    R14, CREATE.ARG_PTR(R15)
00EA 0000
00EC 61E3      LD    R3, ARG_LIST.USR_STK(R14)
00EE 0024
               ! SAVE LIMITS !
00F0 6FF3      LD    CREATE.LIMITS(R15), R3
00F2 0004
```

```
00F4 5F00    CALL  MM_ALLOCATE !R3: # OF BLOCKS
00F6 0000*
                                RETURNS:
                                R2: START ADDR!
             !COMPUTE & SAVE NSP!
00F8 A128    LD    R8, R2
             ! ESTABLISH INITIAL SP VALUE
               FOR USER STACK. !
00FA 0108    ADD   R8, #STK_OFFSET
00FC 00FF
00FE 6FF8    LD    CREATE.N_S_P(R15), R8
0100 0008
             ! RESTORE LIMITS !
0102 61F4    LD    R4, CREATE.LIMITS(R15)
0104 0004
0106 AB40    DEC   R4  !SEG LIMITS!
             ! RESTORE DBR !
0108 61F0    LD    R0, CREATE.DBR_NUM(R15)
010A 0002
010C 2101    LD    R1, #USER_STACK
010E 0003
0110 2103    LD    R3, #WRITE  !ATTRIBUTE!
0112 0000
0114 5F00    CALL  UPDATE_MMU_IMAGE
0116 0000*
                   !R0: DBR #
                    R1: SEGMENT #
                    R2: SEG ADDRESS
                    R3: SEG ATTRIBUTES
                    R4: SEG LIMITS!
             !CREATE KERNEL STACK!
             ! RESTORE ARGUMENT POINTER !
0118 61FE    LD    R14, CREATE.ARG_PTR(R15)
011A 0000
011C 61E3    LD    R3, ARG_LIST.KER_STK(R14)
011E 0026
0120 5F00    CALL  MM_ALLOCATE !R3: # OF BLOCKS
0122 0000*
                                RETURNS
                                R2: START ADDR!
             !MAKE MMU ENTRY!
             ! RESTORE DBR # !
0124 61F0    LD    R0, CREATE.DBR_NUM(R15)
0126 0002
0128 2101    LD    R1, #KERNEL_STACK
012A 0001
012C A134    LD    R4, R3
012E AB40    DEC   R4
0130 2103    LD    R3, #WRITE
0132 0000
             ! SAVE START ADDRESS !
```

```
0134 6FF2    LD     CREATE.SEG_ADDR(R15), R2
0136 0006
0138 5F00    CALL   UPDATE_MMU_IMAGE
013A 0000*
                    !R0: DBR #
                     R1: SEGMENT #
                     R2: SEG ADDRESS
                     R3: SEG ATTRIBUTES
                     R4: SEG LIMITS!
             !ESTABLISH ARGUMENTS!
             ! RESTORE ARGUMENT POINTER !
013C 61FE    LD     R14, CREATE.ARG_PTR(R15)
013E 0000
             ! RESTORE STACK ADDRESS !
0140 61F1    LD     R1, CREATE.SEG_ADDR(R15)
0142 0006
0144 2103    LD     R3, #USER_FCW
0146 1800
0148 61E4    LD     R4, ARG_LIST.IC(R14)
014A 001A
             ! RESTORE INITIAL NSP !
014C 61F5    LD     R5, CREATE.N_S_P(R15)
014E 0008
0150 7606    LDA    R6, VIRT_PREEMPT_RETURN
0152 0076'
0154 030F    SUB    R15, #8
0156 0008
0158 1CF9    LDM    @R15, R3, #4
015A 0303
             ! LOAD ARGUMENT POINTER FOR
               CREATE_STACK CALL !
015C A1F0    LD     R0, R15
015E 93F1    PUSH   @R15, R1
0160 A1E1    LD     R1, R14
             ! LOAD INITIAL REGISTER VALUES TO
               BE PASSED TO USER PROCESS AS
               INITIAL PARAMETERS. !
0162 5C11    LDM    R2, ARG_LIST.REG(R1), #13
0164 020C
0166 0000
0168 97F1    POP    R1, @R15
016A 5F00    CALL   CREATE_STACK
016C 0000*
                    !R0: ARGUMENT PTR
                     R1: TOP OF STACK
                     R2-R14: INITIAL
                      REG STATES!
             !NOTE: THE ABOVE INITIAL REG STATES
              REPRESENT THE INITIAL PARAMETERS
              (VIZ., REGISTER CONTENTS) THAT A
              USER PROCESS WILL RECEIVE UPON
```

```
                     INITIAL EXECUTION. !
016E 010F          ADD   R15, #8  !OVERLAY PARAMETERS!
0170 0008
                   ! ALLOCATE KST !
0172 2103          LD    R3, #KST_LIMIT
0174 0001
0176 5F00          CALL  MM_ALLOCATE !R3:# OF BLOCKS
0178 0000*
                                      RETURNS
                                      R2:START ADDR!
                   ! RESTORE DBR !
017A 61F0          LD    R0, CREATE.DBR_NUM(R15)
017C 0002
                   ! SAVE KST ADDRESS !
017E 6FF2          LD    CREATE.SEG_ADDR(R15), R2
0180 0006
                   !MAKE MMU ENTRY FOR KST SEG!
0182 2101          LD    R1, #KST_SEG
0184 0002
0186 2103          LD    R3, #WRITE !ATTRIBUTE!
0188 0000
018A 2104          LD    R4, #KST_LIMIT-1
018C 0000
018E 5F00          CALL  UPDATE_MMU_IMAGE
0190 0000*
                         !R0: DBR #
                          R1: SEGMENT #
                          R2: SEG ADDRESS
                          R3: SEG ATTRIBUTES
                          R4: SEG LIMITS!
                   ! RESTORE KST ADDRESS !
0192 61F2          LD    R2, CREATE.SEG_ADDR(R15)
0194 0006
                   ! CREATE INITIAL KST STUB !
0196 5F00          CALL  CREATE_KST !R2:KST ADDR!
0198 01A0'
                   ! REMOVE TEMPORARY VARIABLE
                     STACK FRAME. !
019A 010F          ADD   R15, #SIZEOF CREATE
019C 000A
019E 9E08          RET
01A0              END CREATE_PROCESS
```

```
01A0         CREATE_KST     PROCEDURE
            !************************
             * CREATES KST STUB FOR *
             * PROCESS MANAGEMENT   *
             * DEMO.  INSERTS ROOT  *
             * ENTRY IN KST.  NOT   *
             * INTENDED TO BE FINAL *
             * PRODUCT.             *
             ************************
             * PARAMETERS:          *
             *  R2: KST ADDRESS     *
             ************************!

             ENTRY
              !NOTE: THIS PROCEDURE IS A STUB USED
               FOR INITIALIZATION IN THIS IMPLEMENTATION
               ONLY.  THE ACTUAL INITIALIZATION CODE
               FOR THE KST WILL RESIDE AT THE SEGMENT
               MANAGER LEVEL ONCE IMPLEMENTATION OF
               SYSTEM INITIALIZATION IS EFFECTED. !

              ! CREATE ROOT ENTRY IN KST !
01A0 1406     LDL   RR6, #-1  !ROOT HANDLE!
01A2 FFFF
01A4 FFFF
01A6 5D26     LDL   KST.MM_HANDLE(R2), RR6
01A8 0000
              !SET ROOT ENTRY # IN G_AST !
01AA 4D25     LD    KST.MM_HANDLE[2](R2), #0
01AC 0004
01AE 0000
              ! SET ROOT CLASSIFICATION !
01B0 1406     LDL   RR6, #SYSTEM_LOW
01B2 0000
01B4 0000
01B6 5D26     LDL   KST.CLASS(R2), RR6
01B8 000A
              !SET MENTOR SEG #!
01BA 4C25     LDB   KST.M_SEG_NO(R2), #0
01BC 000E
01BE 0000
              !INITIALIZE FREE KST ENTRIES
               FOR DEMO. NOT FULL KST!
01C0 2101     LD    R1, #10
01C2 000A
              DO
01C4 0B01      CP   R1, #0
01C6 0000
01C8 5E0E      IF EQ THEN EXIT FI
01CA 01D0'
01CC 5E08
```

```
01CE 01DE'
01D0 0102       ADD  R2, #SIZEOF KST_REC
01D2 0010
01D4 4C25       LDB  KST.M_SEG_NO(R2), #%FF
01D6 000E
01D8 FFFF
01DA AB10       DEC  R1
01DC E8F3      OD
01DE 9E08      RET
01E0          END CREATE_KST
```

```
Ø1EØ          TC_ADVANCE                 PROCEDURE
              !*********************************
              * EVENTCOUNT IS ADVANCED BY      *
              * INVOCATION OF MM_ADVANCE.      *
              * PROCESSES THAT ARE AWAITING    *
              * THIS EVENT OCCURRENCE ARE      *
              * REMOVED FROM THE BLOCKED LIST*
              * AND MADE READY.  THE READY     *
              * LISTS ARE THEN CHECKED TO      *
              * INSURE PROPER SHEDULING IS     *
              * EFFECTED.  IF NECESSARY VIR- *
              * TUAL PREEMPTS ARE SENT TO ALL*
              * THOSE VP'S BOUND TO LOWER      *
              * PRIORITY PROCESSES.            *
              *********************************
              * PARAMETERS:                    *
              *  R1: HANDLE POINTER            *
              *  R2: INSTANCE (EVENT #)        *
              *********************************
              * RETURNS:                       *
              *  RØ: SUCCESS CODE              *
              *********************************!

              ENTRY
               ! ESTABLISH TEMPORARY VARIABLE
                 STACK FRAME. !
Ø1EØ Ø30F      SUB    R15, #SIZEOF TEMP
Ø1E2 ØØ12
               ! SAVE INPUT ARGUMENTS !
Ø1E4 6FF1      LD     TEMP.HANDLE_PTR(R15), R1
Ø1E6 ØØØØ
Ø1E8 6FF2      LD     TEMP.EVENT_NR(R15), R2
Ø1EA ØØØ2
               ! LOCK APT !
Ø1EC 7604      LDA    R4, APT.LOCK
Ø1EE ØØØØ'
Ø1FØ 5FØØ      CALL   K_LOCK
Ø1F2 ØØØØ*
               ! RETURNS WHEN APT IS LOCKED !
               ! ANNOUNCE EVENT OCCURRENCE BY
                 INCREMENTING EVENTCOUNT IN G_AST!
Ø1F4 5FØØ      CALL   MM_ADVANCE !R1:HANDLE PTR
Ø1F6 ØØØØ*
                                  R2:INSTANCE
                                  RETURNS:
                                  RØ:SUCCESS CODE
                                  RR2:EVENTCOUNT!
Ø1F8 ØBØØ      CP     R2, #SUCCEEDED
Ø1FA ØØØ2
Ø1FC 5EØE      IF EQ THEN
Ø1FE Ø372'
```

```
                 ! SAVE EVENTCOUNT !
0200 5DF2        LDL    TEMP.EVENT_VAL(R15), RR2
0202 0004
                 ! RESTORE INSTANCE !
0204 61F0        LD     R0, TEMP.EVENT_NR(R15)
0206 0002
                 ! RESTORE HANDLE POINTER !
0208 61F1        LD     R1, TEMP.HANDLE_PTR(R15)
020A 0000
                 ! SAVE HANDLE !
020C 5414        LDL    RR4, HANDLE_VAL.HIGH(R1)
020E 0000
0210 5DF4        LDL    TEMP.HANDLE_HIGH(R15), RR4
0212 000C
0214 6114        LD     R4, HANDLE_VAL.LOW(R1)
0216 0004
0218 6FF4        LD     TEMP.HANDLE_LOW(R15), R4
021A 0010
                 ! AWAKEN ALL PROCESSES AWAITING
                   THIS EVENT OCCURRENCE !
                 ! GET FIRST BLOCKED PROCESS !
021C 6101        LD     R1, APT.BLOCKED_LIST
021E 000A'
0220 7606        LDA    R6, APT.BLOCKED_LIST
0222 000A'
                WAKE_UP:
                 DO
                  ! DETERMINE IF AT END OF BLOCKED LIST !
0224 0B01         CP     R1, #NIL
0226 FFFF
                  IF EQ ! NO MORE BLOCKED PROCESSES !
0228 5E0E          THEN EXIT FROM WAKE_UP
022A 0230'
022C 5E08
022E 02B4'
                  FI
                  ! SAVE NEXT ITEM IN LIST !
0230 6117         LD     R7, APT.AP.NEXT_AP(R1)
0232 0020'
                  ! DETERMINE IF PROCESS IS ASSOCIATED
                    WITH CURRENT HANDLE !
0234 54F4         LDL    RR4, TEMP.HANDLE_HIGH(R15)
0236 000C
0238 5014         CPL    RR4, APT.AP.HANDLE(R1)
023A 0030'
                  IF EQ !HIGH HANDLE VALUE MATCHES!
023C 5E0E         THEN
023E 02A2'
0240 61F4         LD     R4, TEMP.HANDLE_LOW(R15)
0242 0010
0244 4B14         CP     R4, APT.AP.HANDLE[2](R1)
```

```
0246 0034'
                    IF EQ ! HANDLE'S MATCH !
0248 5E0E            THEN ! CHECK FOR INSTANCE MATCH !
024A 029C'
024C 61F0             LD    R0, TEMP.EVENT_NR(R15)
024E 0002
0250 4B10             CP    R0, APT.AP.INSTANCE(R1)
0252 0036'
                      IF EQ ! INSTANCE MATCHES !
0254 5E0E              THEN !DETERMINE IF THIS IS THE
0256 0296'
                             OCCURRENCE THE PROCESS
                             WAITING FOR !
0258 54F2               LDL   RR2, TEMP.EVENT_VAL(R15)
025A 0004
025C 5012               CPL   RR2, APT.AP.VALUE(R1)
025E 0038'
                        IF GE !AWAITED EVENT HAS OCCURRED!
0260 5E01                THEN ! AWAKEN PROCESS !
0262 0290'
                          ! REMOVE FROM BLOCKED LIST !
0264 2F67                 LD    @R6, R7
                          ! SAVE LOCAL VARIABLES !
0266 91F6                 PUSHL @R15, RR6
                          !SET LIST THREADING ARGUMENTS!
0268 6112                 LD    R2, APT.AP.AFFINITY(R1)
026A 002C'
026C 7623                 LDA   R3, APT.READY_LIST(R2)
026E 0006'
0270 7604                 LDA   R4, APT.AP.NEXT_AP
0272 0020'
0274 7605                 LDA   R5, APT.AP.PRI
0276 0028'
0278 7606                 LDA   R6, APT.AP.STATE
027A 002A'
027C 2107                 LD    R7, #READY
027E 0001
0280 A112                 LD    R2, R1
0282 5F00                 CALL  LIST_INSERT
0284 0000*
                           !R2: OBJ ID
                            R3: LIST HEAD PTR
                            R4: NEXT OBJ PTR
                            R5: PRIORITY PTR
                            R6: STATE PTR
                            R7: STATE VALUE !
                          ! RESTORE LOCAL VARIABLES !
0286 95F6                 POPL  RR6, @R15
0288 210B                 LD    R11, #REMOVED
028A ABCD
028C 5E08                ELSE !PROCESS STILL BLOCKED!
```

```
028E 0292'
0290 8DB8            CLR   R11
                   FI ! END VALUE CHECK !
0292 5E08         ELSE !PROCESS STILL BLOCKED!
0294 0298'
0296 8DB8          CLR   R11
                 FI ! END INSTANCE CHECK !
0298 5E08       ELSE  !PROCESS STILL BLOCKED!
029A 029E'
029C 8DB8         CLR   R11
               FI ! END HANDLE CHECK !
029E 5E08      ELSE  !PROCESS STILL BLOCKED!
02A0 02A4'
02A2 8DB8       CLR   R11
               FI ! END HIGH HANDLE CHECK !
               ! RESET AP POINTER REGISTERS !
02A4 0B0B      CP    R11, #REMOVED
02A6 ABCD
               IF NE ! PROCESS IS STILL BLOCKED !
02A8 5E06       THEN
02AA 02B0'
02AC 7616        LDA   R6, APT.AP.NEXT_AP(R1)
02AE 0020'
               FI
02B0 A171      LD    R1, R7
02B2 E8B8     OD
              ! DETERMINE IF ANY VIRTUAL PREEMPT
                INTERRUPTS ARE REQUIRED !
02B4 8D28     CLR   R2
             PREEMPT_CHECK:
              DO
02B6 0B02      CP    R2, #NR_CPU * 2
02B8 0004
02BA 5E0E      IF EQ !ALL READY LISTS CHECKED! THEN
02BC 02C2'
02BE 5E08       EXIT FROM PREEMPT_CHECK
02C0 0366'
               FI
               ! CREATE PREEMPT VECTOR FOR VP'S !
02C2 8D18      CLR   R1
               DO !FOR R1=1 TO NR_VP'S!
02C4 A910       INC   R1
02C6 4B21       CP    R1, APT.VP.NR_VP(R2)
02C8 0010'
                IF GT ! PREEMPT VECTOR COMPLETED !
02CA 5E02        THEN EXIT
02CC 02D2'
02CE 5E08
02D0 02D8'
                FI
02D2 0DF9       PUSH  @R15, #TRUE
```

```
02D4 0001
02D6 E6F6      OD
               ! # TO PREEMPT !
02D8 8D38      CLR    R3
02DA 6124      LD     R4, APT.VP.NR_VP(R2)
02DC 0010'
               ! # OF VP'S !
               ! GET FIRST READY PROCESS !
02DE 6121      LD     R1, APT.READY_LIST(R2)
02E0 0006'
              CHECK_RDY_LIST:
               DO
                ! SEE IF READY LIST IS EMPTY !
02E2 0B01       CP     R1, #NIL
02E4 FFFF
                IF EQ !LIST IS EMPTY!
02E6 5E0E        THEN EXIT FROM CHECK_RDY_LIST
02E8 02EE'
02EA 5E08
02EC 0324'
                FI
02EE 4D11       CP     APT.AP.STATE(R1), #RUNNING
02F0 002A'
02F2 0000
                IF EQ !PROCESS IS RUNNING!
02F4 5E0E        THEN !DON'T PREEMPT IT!
02F6 030C'
02F8 6115          LD     R5, APT.AP.VP_ID(R1)
02FA 002E'
                  !COMPUTE LOCATION IN PREEMPT VECTOR!
02FC 4325         SUB    R5, APT.VP.FIRST(R2)
02FE 0014'
0300 74F6         LDA    R6, R15(R5)
0302 0500
0304 0D65         LD     @R6, #FALSE
0306 0000
0308 5E08        ELSE ! PREEMPT IT !
030A 030E'
030C A930         INC    R3
                FI
030E AB40       DEC    R4
0310 0B04       CP     R4, #0
0312 0000
                IF EQ  !ALL VP'S VERIFIED!
0314 5E0E        THEN
0316 031C'
0318 5E08         EXIT FROM CHECK_RDY_LIST
031A 0324'
                FI
                ! GET NEXT AP IN READY LIST !
031C 6110       LD     R0, APT.AP.NEXT_AP(R1)
```

```
031E 0020'
0320 A101       LD    R1, R0
0322 E8DF      OD  !END CHECK_RDY_LIST!
               ! SET NECESSARY PREEMPTS !
0324 6124      LD    R4, APT.VP.NR_VP(R2)
0326 0010'
0328 6121      LD    R1, APT.VP.FIRST(R2)
032A 0014'
              SEND_PREEMPT:
               DO
032C 97F0       POP   R0, @R15
                ! CHECK TEMPLATE !
032E 0B00       CP    R0, #TRUE
0330 0001
                IF EQ !CAN BE PREEMPTED!
0332 5E0E        THEN
0334 0350'
0336 0B03         CP    R3, #0
0338 0000
                  IF GT !PREEMPTS REQUIRED!
033A 5E02          THEN !PREEMPT IT!
033C 0350'
                    !SAVE ARGUMENTS!
033E 93F1           PUSH  @R15, R1
0340 91F2           PUSHL @R15, RR2
0342 93F4           PUSH  @R15, R4
0344 5F00           CALL  SET_PREEMPT
0346 0000*
                     !P1: VP ID!
                    ! RESTORE ARGUMENTS !
0348 97F4           POP   R4, @R15
034A 95F2           POPL  RR2, @R15
034C 97F1           POP   R1, @R15
034E AB30           DEC   R3
                  FI
                FI
0350 A911       INC   R1, #2
0352 AB40       DEC   R4
0354 0B04       CP    R4, #0
0356 0000
                IF EQ !STACK RESTORED!
0358 5E0E        THEN
035A 0360'
035C 5E08         EXIT
035E 0362'
                FI
0360 E8E5      OD !END SEND_PREEMPT!
               ! CHECK NEXT READY LIST !
0362 A921      INC   R2, #2
0364 E8A8     OD !END PREEMPT_CHECK!
```

```
                    ! UNLOCK APT !
0366 7604           LDA    R4, APT.LOCK
0368 0000'
036A 5F00           CALL   K_UNLOCK
036C 0000*
                    ! RESTORE SUCCESS CODE !
036E 2100           LD     R0, #SUCCEEDED
0370 0002
                    FI
                    ! RESTORE STACK !
0372 010F           ADD    R15, #SIZEOF TEMP
0374 0012
0376 9E08           RET
0378              END TC_ADVANCE
```

```
0378          TC_AWAIT                     PROCEDURE
             !********************************
              * CHECKS USER SPECIFIED VALUE   *
              * AGAINST CURRENT EVENTCOUNT    *
              * VALUE. IF USER VALUE IS LESS  *
              * THAN OR EQUAL EVENTCOUNT THEN*
              * CONTROL IS RETURNED TO USER. *
              * ELSE USER IS BLOCKED UNTIL    *
              * EVENT OCCURRENCE.             *
              ********************************
              * PARAMETERS:                   *
              *  R1: HANDLE POINTER           *
              *  R2: INSTANCE (EVENT #)       *
              *  RR4: SPECIFIED VALUE         *
              ********************************
              * RETURNS:                      *
              *  R0: SUCCESS CODE             *
              ********************************!

              ENTRY
               ! ESTABLISH STACK FRAME FOR
                 TEMPORARY VARIABLES. !
0378 030F      SUB   R15, #SIZEOF TEMP
037A 0012
               ! SAVE INPUT PARAMETERS !
037C 6FF1      LD    TEMP.HANDLE_PTR(R15), R1
037E 0000
0380 6FF2      LD    TEMP.EVENT_NR(R15), R2
0382 0002
0384 5DF4      LDL   TEMP.EVENT_VAL(R15), RR4
0386 0004
               ! LOCK APT !
0388 7604      LDA   R4, APT.LOCK
038A 0000'
038C 5F00      CALL  K_LOCK
038E 0000*
               ! RETURNS WHEN APT IS LOCKED !
               ! GET CURRENT EVENTCOUNT !
0390 5F00      CALL  MM_READ_EVENTCOUNT
0392 0000*
                     !R1:HANDLE POINTER
                      R2:INSTANCE
                     RETURNS:
                      R0:SUCCESS_CODE
                      RR4: EVENTCOUNT!
0394 0B00      CP    R0, #SUCCEEDED
0396 0002
0398 5E0E      IF EQ THEN
039A 0440'
               ! DETERMINE IF REQUESTED EVENT
                 HAS OCCURRED !
```

```
039C 54F6    LDL   RR6, TEMP.EVENT_VAL(R15)
039E 0004
03A0 9046    CPL   RR6, RR4
             IF  GT  !EVENT HAS NOT OCCURRED!
03A2 5E02     THEN  !BLOCK PROCESS!
03A4 0440'
               ! IDENTIFY PROCESS !
03A6 5F00      CALL  RUNNING_VP  !RETURNS:
03A8 0000*
                                   R1:VP ID
                                   R3:CPU #!
               ! SAVE RETURN VARIABLES !
03AA 6FF1      LD     TEMP.ID_VP(R15), R1
03AC 0008
03AE 6FF3      LD     TEMP.CPU_NUM(R15), R3
03B0 000A
03B2 6118      LD     R8, APT.RUNNING_LIST(R1)
03B4 0002'
               ! RESTORE REMAINING ARGUMENTS !
03B6 61F2      LD     R2, TEMP.EVENT_NR(R15)
03B8 0002
03BA 61F1      LD     R1, TEMP.HANDLE_PTR(R15)
03BC 0000
               ! SAVE EVENT DATA !
03BE 5414      LDL   RR4, HANDLE_VAL.HIGH(R1)
03C0 0000
03C2 5D84      LDL   APT.AP.HANDLE(R8), RR4
03C4 0030'
03C6 6114      LD     R4, HANDLE_VAL.LOW(R1)
03C8 0004
03CA 6F84      LD     APT.AP.HANDLE[2](R8), R4
03CC 0034'
03CE 6F82      LD     APT.AP.INSTANCE(R8), R2
03D0 0036'
03D2 54F6      LDL   RR6, TEMP.EVENT_VAL(R15)
03D4 0004
03D6 5D86      LDL   APT.AP.VALUE(R8), RR6
03D8 0038'
               ! REMOVE PROCESS FROM READY LIST !
03DA 6181      LD     R1, APT.AP.AFFINITY(R8)
03DC 002C'
03DE 6112      LD     R2, APT.READY_LIST(R1)
03E0 0006'
               ! SEE IF PROCESS IS FIRST
                 ENTRY IN READY LIST !
03E2 8B82      CP     R2, R8
               IF EQ !INSERT NEW READY LIST HEAD!
03E4 5E0E       THEN
03E6 03F4'
03E8 6183        LD     R3, APT.AP.NEXT_AP(R8)
03EA 0020'
```

```
03EC 6F13        LD    APT.READY_LIST(R1), R3
03EE 0006'
03F0 5E08       ELSE !DELETE FROM LIST BODY!
03F2 040E'
                 DO
03F4 6123         LD    R3, APT.AP.NEXT_AP(R2)
03F6 0020'
03F8 8B83         CP    R3, R8
                  IF EQ !FOUND ITEM IN LIST!
03FA 5E0E          THEN
03FC 040A'
03FE 6183           LD    R3, APT.AP.NEXT_AP(R8)
0400 0020'
0402 6F23           LD    APT.AP.NEXT_AP(R2), R3
0404 0020'
0406 5E08           EXIT
0408 040E'
                  FI
040A A132         LD    R2, R3
040C E8F3        OD
               FI
               !THREAD PROCESS IN BLOCKED LIST!
040E A182      LD    R2, R8
0410 7603      LDA   R3, APT.BLOCKED_LIST
0412 000A'
0414 7604      LDA   R4, APT.AP.NEXT_AP
0416 0020'
0418 7605      LDA   R5, APT.AP.PRI
041A 0028'
041C 7606      LDA   R6, APT.AP.STATE
041E 002A'
0420 2107      LD    R7, #BLOCKED
0422 0002
0424 5F00      CALL  LIST_INSERT !R2:OBJ ID
0426 0000*
                                 R3:LIST HEAD PTR
                                 R4:NEXT OBJ PTR
                                 R5:PRIORITY PTR
                                 R6:STATE PTR
                                 R7:STATE !
               ! GET CURRENT VP ID !
0428 61F1      LD    R1, TEMP.ID_VP(R15)
042A 0008
042C 61F3      LD    R3, TEMP.CPU_NUM(R15)
042E 000A
               ! SCHEDULE FIRST READY PROCESS !
0430 5F00      CALL  TC_GETWORK  !R1:VP_ID
0432 0000'
                                 R3:CPU #!
               ! UNLOCK APT !
0434 7604      LDA   R4, APT.LOCK
```

```
0436 0000'
0438 5F00         CALL  K_UNLOCK
043A 0000*
                  ! RESTORE SUCCESS CODE !
043C 2100         LD    R0, #SUCCEEDED
043E 0002
               FI
              FI
               ! RESTORE STACK !
0440 010F      ADD   R15, #SIZEOF TEMP
0442 0012
0444 9E08      RET
0446          END TC_AWAIT
```

```
0446          PROCESS_CLASS       PROCEDURE
             !****************************
              * READS SECURITY ACCESS     *
              * CLASS OF CURRENT PROCESS  *
              * IN APT.  CALLED BY SEG    *
              * MGR AND EVENT MGR         *
              ****************************
              * LOCAL VARIABLES:          *
              *  R1: VP ID                *
              *  R5: PROCESS ID           *
              ****************************
              * RETURNS:                  *
              *  RR2: PROCESS SAC         *
              ****************************!

              ENTRY
0446 7604      LDA   R4,APT.LOCK
0448 0000'
044A 5F00      CALL  K_LOCK    !R4:^APT.LOCK!
044C 0000*
044E 5F00      CALL  RUNNING_VP  !RETURNS:
0450 0000*
                                  R1:VP_ID
                                  R3:CPU #!
0452 6115      LD    R5,APT.RUNNING_LIST(R1)
0454 0002'
0456 5452      LDL   RR2,APT.AP.SAC(R5)
0458 0024'
               ! UNLOCK APT !
045A 7604      LDA   R4, APT.LOCK
045C 0000'
045E 5F00      CALL  K_UNLOCK
0460 0000*
0462 9E08     RET
0464          END PROCESS_CLASS
```

```
0464          GET_DBR_NUMBER          PROCEDURE
             !*********************************
              * OBTAINS DBR NUMBER FROM APT   *
              * FOR THE CURRENT PROCESS.      *
              * CALLED BY SEGMENT MANAGER     *
              *********************************
              * LOCAL VARIABLES:              *
              *  R1: VP ID                    *
              *  R5: PROCESS ID               *
              *********************************
              * RETURNS:                      *
              *  R1: DBR NUMBER               *
              *********************************!

              ENTRY
               !NOTE: DBR # IS ONLY VALID WHILE PROCESS
                IS LOADED.  THIS IS NO PROBLEM IN SASS
                AS ALL PROCESSES REMAIN LOADED.  IN A
                MORE GENERAL CASE, THE DBR # COULD ONLY
                BE ASSUMED CORRECT WHILE THE APT IS LOCKED!
0464 7604      LDA    R4,APT.LOCK
0466 0000'
0468 5F00      CALL   K_LOCK   !R4:^APT.LOCK!
046A 0000*
046C 5F00      CALL   RUNNING_VP  !RETURNS:
046E 0000*
                                   R1:VP_ID
                                   R3:CPU #!
0470 6115      LD     R5,APT.RUNNING_LIST(R1)
0472 0002'
0474 6151      LD     R1,APT.AP.DBR(R5)
0476 0022'
               ! UNLOCK APT !
0478 7604      LDA    R4, APT.LOCK
047A 0000'
047C 5F00      CALL   K_UNLOCK
047E 0000*
0480 9E08      RET
0482          END GET_DBR_NUMBER

              END TC
```

```
Z8000ASM  2.02
LOC     OBJ CODE    STMT SOURCE STATEMENT

          $LISTON $TTY
          DIST_MM MODULE

          CONSTANT

          CREATE_CODE              := 50
          DELETE_CODE              := 51
          ACTIVATE_CODE            := 52
          DEACTIVATE_CODE          := 53
          SWAP_IN_CODE             := 54
          SWAP_OUT_CODE            := 55
          NR_CPU                   := 2
          NR_KST_ENTRY             := 54
          MAX_SEG_SIZE             := 128
          MAX_DBR_NO               := 4
          KST_SEG_NO               := 2
          NR_OF_KSEGS              := 10
          BLOCK_SIZE               := 8
          MEM_AVAIL                := %F00
          G_AST_LIMIT              := 10
          INSTANCE1                := 1
          INSTANCE2                := 2
          INVALID_INSTANCE         := 95
          SUCCEEDED                := 2

        TYPE
          H_ARRAY             ARRAY [3   WORD]
          COM_MSG             ARRAY [16  BYTE]
          ADDRESS             WORD

          G_AST_REC         RECORD
           [UNIQUE_ID       LONG
            GLOBAL_ADDR     ADDRESS
            P_L_ASTE_NO     WORD
            FLAG            WORD
            PAR_ASTE        WORD
            NR_ACTIVE       WORD
            NO_ACT_DEP      BYTE
            SIZE1           BYTE
            PG_TBL          ADDRESS
            ALIAS_TBL       ADDRESS
            SEQUENCER       LONG
            EVENT1          LONG
            EVENT2          LONG
           ]
        MM_VP_ID           WORD

        SEG_ARRAY         ARRAY [MAX_SEG_SIZE    BYTE]

        $SECTION D_MM_DATA
        GLOBAL

0000    MM_CPU_TBL ARRAY [NR_CPU MM_VP_ID]

        $SECTION AVAIL_MEM
        INTERNAL
        ! NOTE: MEM_POOL IS LOCATED IN
          CPU LOCAL MEMORY. !
0000    MEM_POOL      ARRAY [MEM_AVAIL BYTE]

      GLOBAL
        ! NOTE: NEXT_BLOCK IS USED IN THE MM_ALLOCATE
          STUB AS AN OFFSET POINTER INTO THE BLOCK
          OF ALLOCATABLE MEMORY.  IT IS INITIALIZED
          IN BOOTSTRAP LOADER. !
0F00    NEXT_BLOCK       WORD
        $SECTION MSG_FRAME_DCL
      INTERNAL
        !NOTE: THESE RECORDS ARE "OVERLAYS" OR "FRAMES" USED
         TO DEFINE MESSAGE FORMATS. NO MEMORY IS ALLOCATED !
        $ABS 0
0000    CREATE_MSG        RECORD [CR_CODE        WORD
                                  CE_MM_HANDLE   H_ARRAY
                                  CE_ENTRY_NO    SHORT_INTEGER
                                  CE_FILL        BYTE
                                  CE_SIZE        WORD
                                  CE_CLASS       LONG]

        $ABS 0
0000    DELETE_MSG        RECORD [DE_CODE        WORD
                                  DE_MM_HANDLE   H_ARRAY
                                  DE_ENTRY_NO    SHORT_INTEGER
                                  DE_FILL        ARRAY[7 BYTE]]

        $ABS 0
0000    ACTIVATE_MSG   RECORD [ACT_CODE       WORD
                                  A_DBR_NO       WORD
                                  A_MM_HANDLE    H_ARRAY
                                  A_ENTRY_NO     SHORT_INTEGER
                                  A_SEGMENT_NO   SHORT_INTEGER
                                  A_FILL         LONG]
        $ABS 0
0000    DEACTIVATE_MSG     RECORD [DEACT_CODE    WORD
                                   D_DBR_NO      WORD
                                   D_MM_HANDLE   H_ARRAY
                                   D_FILL        ARRAY[3 WORD]]

        $ABS 0
0000    SWAP_IN_MSG       RECORD [S_IN_CODE     WORD
                                   SI_MM_HANDLE  H_ARRAY
                                   SI_DBR_NO     WORD
                                   SI_ACCESS_AUTH BYTE
                                   SI_FILL1      BYTE
                                   SI_FILL       ARRAY[2 WORD]]
        $ABS 0
0000    SWAP_OUT_MSG      RECORD [S_OUT_CODE    WORD
                                   SO_DBR_NO     WORD
                                   SO_MM_HANDLE  H_ARRAY
                                   SO_FILL       ARRAY[3 WORD]]

        $ABS 0
0000    RET_SUC_CODE        RECORD [SUC_CODE      BYTE
                                   SC_FILL       ARRAY[15 BYTE]
                                  ]

        $ABS 0
0000    R_ACTIVATE_ARG    RECORD [R_SUC_CODE    BYTE
                                   R_FILL        BYTE
                                   R_MM_HANDLE   H_ARRAY
                                   R_CLASS       LONG
                                   R_SIZE        WORD
                                   R_FILL1       WORD]

        $ABS 0
0000    MM_HANDLE      RECORD
          [ID       LONG
           ENTRY_NO WORD
          ]
```

```
EXTERNAL
        G_AST_LOCK          WORD
        G_AST  ARRAY[G_AST_LIMIT G_AST_REC]
        K_LOCK              PROCEDURE
        K_UNLOCK            PROCEDURE
        GET_CPU_NO     PROCEDURE
        SIGNAL              PROCEDURE
        WAIT                PROCEDURE
```

```
          GLOBAL
          $SECTION D_MM_PROC

0000              MM_CREATE_ENTRY          PROCEDURE
                 !*********************************
                  * INTERFACE BETWEEN SEG MGR     *
                  * (CREATE_SEG PROCEDURE) AND    *
                  * MMGR PROCESS (CREATE_ENTRY    *
                  * PROCEDURE). ARRANGES AND      *
                  * PERFORMS IPC.                 *
                  *********************************
                  * REGISTER USE:                 *
                  * PARAMETERS                    *
                  *  R0:SUCCESS_CODE (RET)        *
                  *  R1:HPTR (INPUT)              *
                  *  R2:ENTRY_NO (INPUT)          *
                  *  R3:SIZE (INPUT)              *
                  *  RR4:CLASS (INPUT)            *
                  * LOCAL USE                     *
                  *  R6:MM_HANDLE ARRAY ENTRY     *
                  *  R8:^COM_MSGBUF               *
                  *  R13:^COM_MSGBUF              *
                  *********************************!
                  ENTRY
                   !USE STACK FOR MESSAGE!
0000 030F          SUB   R15,#SIZEOF COM_MSG
0002 0010
0004 A1FD          LD    R13,R15   ! ^COM_MSGBUF !

                   !FILL COM_MSGBUF (LOAD MESSAGE). CREATE MSG
                    FRAME IS BASED AT ADDRESS ZERO.  IT IS
                    OVERLAID ONTO COM_MSGBUF FRAME BY INDEXING
                    EACH ENTRY (I.E. ADDING TO EACH ENTRY) THE
                    BASE ADDRESS OF COM_MSGBUF!

0006 4DD5          LD    CREATE_MSG.CR_CODE(R13),#CREATE_CODE
0008 0000
000A 0032
000C 3116          LD    R6,R1(#0)  !INDEX TO MM_HANDLE ENTRY!
000E 0000
0010 6FD6          LD    CREATE_MSG.CE_MM_HANDLE[0](R13),R6
0012 0002
0014 3116          LD    R6,R1(#2)
0016 0002
0018 6FD6          LD    CREATE_MSG.CE_MM_HANDLE[1](R13),R6
001A 0004
001C 3116          LD    R6,R1(#4)
001E 0004
0020 6FD6          LD    CREATE_MSG.CE_MM_HANDLE[2](R13),R6
0022 0006
0024 6FD2          LD    CREATE_MSG.CE_ENTRY_NO(R13),R2
```

```
ØØ26 ØØØ8
ØØ28 5DD4     LDL    CREATE_MSG.CE_CLASS(R13),RR4
ØØ2A ØØØC
ØØ2C 6FD3     LD     CREATE_MSG.CE_SIZE(R13),R3
ØØ2E ØØØA
ØØ3Ø A1D8     LD     R8,R13
ØØ32 5FØØ     CALL   PERFORM_IPC  !R8: ^COM_MSGBUF!
ØØ34 Ø18C
              !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
ØØ36 8DØ8     CLR    RØ
ØØ38 6ØD8     LDB    RLØ,RET_SUC_CODE.SUC_CODE(R13)
ØØ3A ØØØØ
ØØ3C Ø1ØF     ADD    R15,#SIZEOF COM_MSG  !RESTORE STACK STATE!
ØØ3E ØØ1Ø
ØØ4Ø 9EØ8     RET
ØØ42       END MM_CREATE_ENTRY
```

```
0042            MM_DELETE_ENTRY           PROCEDURE
               !********************************
                * INTERFACE BETWEEN SEG MGR     *
                * (DELETE_SEG PROCEDURE) AND    *
                * MMGR (DELETE_ENTRY PROCEDURE).*
                * ARRANGES AND PERFORMS IPC.    *
                *********************************
                * REGISTER USE:                 *
                * PARAMETERS                    *
                *  R0:SUCCESS_CODE(RET)         *
                *  R1:HPTR(INPUT)               *
                *  R2:ENTRY_NO(INPUT)           *
                * LOCAL USE                     *
                *  R6:MM_HANDLE ARRAY ENTRY     *
                *  R8:^COM_MSGBUF               *
                *  R13:^COM_MSGBUF              *
                *********************************!
                ENTRY
                 !USE STACK FOR MESSAGE!
0042 030F         SUB   R15,#SIZEOF COM_MSG
0044 0010
0046 A1FD         LD    R13,R15   ! ^COM_MSGBUF !
             !FILL COM_MSGBUF (LOAD MESSAGE). DELETE_MSG FRAME
              IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
              COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
              ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
0048 4DD5         LD     DELETE_MSG.DE_CODE(R13),#DELETE_CODE
004A 0000
004C 0033
004E 3116         LD     R6,R1(#0)  !INDEX TO MM_HANDLE ENTRY!
0050 0000
0052 6FD6         LD     DELETE_MSG.DE_MM_HANDLE[0](R13),R6
0054 0002
0056 3116         LD     R6,R1(#2)
0058 0002
005A 6FD6         LD     DELETE_MSG.DE_MM_HANDLE[1](R13),R6
005C 0004
005E 3116         LD     R6,R1(#4)
0060 0004
0062 6FD6         LD     DELETE_MSG.DE_MM_HANDLE[2](R13),R6
0064 0006
0066 6FD2         LD     DELETE_MSG.DE_ENTRY_NO(R13),R2
0068 0008
006A A1D8         LD     R8,R13
006C 5F00         CALL   PERFORM_IPC  !R8: ^COM_MSGBUF!
006E 018C
                 !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0070 8D08         CLR    R0
0072 60D8         LDB    RL0,RET_SUC_CODE.SUC_CODE(R13)
0074 0000
0076 010F         ADD    R15,#SIZEOF COM_MSG  !RESTORE STACK STATE!
0078 0010
007A 9E08         RET
007C         END MM_DELETE_ENTRY
```

```
007C           MM_ACTIVATE                 PROCEDURE
             !********************************
              * INTERFACE BETWEEN SEG MGR      *
              * (MAKE_KNOWN PROCEDURE) AND     *
              * MMGR  (ACTIVATE PROCEDURE).    *
              * ARRANGES AND PERFORMS IPC.     *
              *********************************
              * REGISTER USE:                  *
              * PARAMETERS                     *
              *  R1:DBR_NO(INPUT)              *
              *  R2:HPTR(INPUT)                *
              *  R3:ENTRY_NO                   *
              *  R4:SEGMENT_NO                 *
              *  R12:RET_HANDLE_PTR            *
              * LOCAL USE                      *
              *  R8:^COM_MSGBUF                *
              *  R13:^COM_MSGBUF               *
              * RETURNS:                       *
              *  R0:SUCCESS CODE               *
              *  RR2:CLASS                     *
              *  R4:SIZE                       *
              *********************************!

              ENTRY
               !USE STACK FOR MESSAGE!
007C 030F      SUB    R15,#SIZEOF COM_MSG
007E 0010
0080 A1FD      LD     R13,R15    ! ^COM_MSGBUF !
               ! SAVE RETURN HANDLE POINTER !
0082 93FC      PUSH   @R15, R12

            !FILL COM_MSGBUF (LOAD MESSAGE). ACTIVATE_MSG FRAME
             IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
             COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
             ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!
0084 4DD5      LD     ACTIVATE_MSG.ACT_CODE(R13),#ACTIVATE_CODE
0086 0000
0088 0034
008A 6FD1      LD     ACTIVATE_MSG.A_DBR_NO(R13),R1
008C 0002
008E 3126      LD     R6,R2(#0)
0090 0000
0092 6FD6      LD     ACTIVATE_MSG.A_MM_HANDLE[0](R13),R6
0094 0004
0096 3126      LD     R6,R2(#2)
0098 0002
009A 6FD6      LD     ACTIVATE_MSG.A_MM_HANDLE[1](R13),R6
009C 0006
009E 3126      LD     R6,R2(#4)
00A0 0004
00A2 6FD6      LD     ACTIVATE_MSG.A_MM_HANDLE[2](R13),R6
```

```
00A4 0008
00A6 6EDB    LDB    ACTIVATE_MSG.A_ENTRY_NO(R13),RL3
00A8 000A
00AA 6EDC    LDB    ACTIVATE_MSG.A_SEGMENT_NO(R13),RL4
00AC 000B
00AE A1D8    LD     R8,R13
00B0 5F00    CALL   PERFORM_IPC !(R8:^COM_MSGBUF!
00B2 018C
             ! RESTORE RETURN HANDLE POINTER !
00B4 97FC    POP    R12, @R15

             ! UPDATE MM_HANDLE ENTRY !
00B6 54D6    LDL    RR6, R_ACTIVATE_ARG.R_MM_HANDLE(R13)
00B8 0002
00BA 5DC6    LDL    MM_HANDLE.ID(R12), RR6
00BC 0000
00BE 61D6    LD     R6,R_ACTIVATE_ARG.R_MM_HANDLE[2](R13)
00C0 0006
00C2 6FC6    LD     MM_HANDLE.ENTRY_NO(R12), R6
00C4 0004
             !RETRIEVE OTHER RETURN ARGUMENTS!
00C6 8D08    CLR    R0
00C8 60D8    LDB    RL0,R_ACTIVATE_ARG.R_SUC_CODE(R13)
00CA 0000
00CC 54D2    LDL    RR2,R_ACTIVATE_ARG.R_CLASS(R13)
00CE 0008
00D0 61D4    LD     R4,R_ACTIVATE_ARG.R_SIZE(R13)
00D2 000C
00D4 010F    ADD    R15,#SIZEOF COM_MSG !RESTORE STACK STATE!
00D6 0010
00D8 9E08    RET
00DA       END MM_ACTIVATE
```

```
00DA          MM DEACTIVATE            PROCEDURE
             !********************************
              * INTERFACE BETWEEN SEG MGR     *
              * (TERMINATE PROCEDURE) AND     *
              * MMGR (DEACTIVATE PROCEDURE).  *
              * ARRANGES AND PERFORMS IPC.    *
              *********************************
              * REGISTER USE:                 *
              * PARAMETERS                    *
              *  R0:SUCCESS_CODE(RET)         *
              *  R1:DBR_NO(INPUT)             *
              *  R2:HPTR(INPUT)               *
              * LOCAL USE                     *
              *  R6:MM_HANDLE ARRAY ENTRY     *
              *  R8:^COM_MSGBUF               *
              *  R13:^COM_MSGBUF              *
              *********************************!

              ENTRY
               !USE STACK FOR MESSAGE!
00DA 030F      SUB   R15,#SIZEOF COM_MSG
00DC 0010
00DE A1FD      LD    R13,R15   ! ^COM_MSGBUF !

           !FILL COM_MSGBUF (LOAD MESSAGE). DEACTIVATE_MSG FRAME
            IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
            COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
            ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

00E0 4DD5      LD    DEACTIVATE_MSG.DEACT_CODE(R13),
00E2 0000                             #DEACTIVATE_CODE
00E4 0035
00E6 6FD1      LD    DEACTIVATE_MSG.D_DBR_NO(R13),R1
00E8 0002
00EA 3126      LD    R6,R2(#0)   !INDEX TO MM_HANDLE ENTRY!
00EC 0000
00EE 6FD6      LD    DEACTIVATE_MSG.D_MM_HANDLE[0](R13),R6
00F0 0004
00F2 3126      LD    R6,R2(#2)
00F4 0002
00F6 6FD6      LD    DEACTIVATE_MSG.D_MM_HANDLE[1](R13),R6
00F8 0006
00FA 3126      LD    R6,R2(#4)
00FC 0004
00FE 6FD6      LD    DEACTIVATE_MSG.D_MM_HANDLE[2](R13),R6
0100 0008
0102 A1D8      LD    R8,R13
0104 5F00      CALL  PERFORM_IPC  !R8: ^COM_MSGBUF!
0106 018C'
```

```
                        !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!

0108 8D08     CLR    R0
010A 60D8     LDB    RL0,RET_SUC_CODE.SUC_CODE(R13)
010C 0000
010E 010F     ADD    R15,#SIZEOF COM_MSG !RESTORE STACK STATE!
0110 0010
0112 9E08     RET
0114       END MM_DEACTIVATE
```

```
0114          MM_SWAP_IN                    PROCEDURE
             !*********************************
              * INTERFACE BETWEEN SEG MGR (SM_*
              * SWAP_IN PROCEDURE) AND MMGR   *
              * (SWAP_IN PROCEDURE). ARRANGES *
              * AND PERFORMS IPC.             *
              *********************************
              * REGISTER USE:                 *
              * PARAMETERS                    *
              *  R0:SUCCESS_CODE(RET)         *
              *  R1:DBR_NO(INPUT)             *
              *  R2:HPTR(INPUT)               *
              *  R3:ACCESS     (INPUT)        *
              * LOCAL USE                     *
              *  R6:MM_HANDLE ARRAY ENTRY     *
              *  R8:^COM_MSGBUF               *
              *  R13:^COM_MSGBUF              *
              *********************************!
              ENTRY
               !USE STACK FOR MESSAGE!
0114 030F      SUB    R15,#SIZEOF COM_MSG
0116 0010
0118 A1FD      LD     R13,R15    ! ^COM_MSGBUF !

            !FILL COM_MSGBUF (LOAD MESSAGE). SWAP_IN_MSG FRAME
             IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
             COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
             ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

011A 4DD5      LD     SWAP_IN_MSG.S_IN_CODE(R13),#SWAP_IN_CODE
011C 0000
011E 0036
0120 3126      LD     R6,R2(#0)  !INDEX TO MM_HANDLE ENTRY!
0122 0000
0124 6FD6      LD     SWAP_IN_MSG.SI_MM_HANDLE[0](R13),R6
0126 0002
0128 3126      LD     R6,R2(#2)
012A 0002
012C 6FD6      LD     SWAP_IN_MSG.SI_MM_HANDLE[1](R13),R6
012E 0004
0130 3126      LD     R6,R2(#4)
0132 0004
0134 6FD6      LD     SWAP_IN_MSG.SI_MM_HANDLE[2](R13),R6
0136 0006
0138 6FD1      LD     SWAP_IN_MSG.SI_DBR_NO(R13),R1
013A 0008
013C 6FDB      LDB    SWAP_IN_MSG.SI_ACCESS_AUTH(R13),RL3
013E 000A
0140 A1D8      LD     R8,R13
0142 5F00      CALL   PERFORM_IPC  !R8: ^COM_MSGBUF!
0144 018C
```

```
                    !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0146 8D08      CLR    R0
0148 60D8      LDB    RL0,RET_SUC_CODE.SUC_CODE(R13)
014A 0000
014C 010F      ADD    R15,#SIZEOF COM_MSG !RESTORE STACK STATE!
014E 0010
0150 9E08      RET
0152        END MM_SWAP_IN
```

```
Ø152          MM_SWAP_OUT                 PROCEDURE
             !*********************************
              * INTERFACE BETWEEN SEG MGR (SM_*
              * SWAP_OUT PROCEDURE) AND MMGR  *
              * (SWAP_OUT PROCEDURE). ARRANGES*
              * AND PERFORMS IPC.             *
              *********************************
              * REGISTER USE:                 *
              * PARAMETERS                    *
              *  RØ:SUCCESS_CODE(RET)         *
              *  R1:DBR_NO(INPUT)             *
              *  R2:HPTR(INPUT)               *
              * LOCAL USE                     *
              *  R6:MM_HANDLE ARRAY ENTRY     *
              *  R8:^COM_MSGBUF               *
              *  R13:^COM_MSGBUF              *
              *********************************!
              ENTRY
               !USE STACK FOR MESSAGE!
Ø152 Ø3ØF      SUB   R15,#SIZEOF COM_MSG
Ø154 ØØ1Ø
Ø156 A1FD      LD    R13,R15    ! ^COM_MSGBUF !

         !FILL COM_MSGBUF (LOAD MESSAGE). SWAP_OUT_MSG FRAME
          IS BASED AT ADDRESS ZERO. IT IS OVERLAID ONTO
          COM_MSGBUF FRAME BY INDEXING EACH ENTRY (I.E. ADD-
          ING TO EACH ENTRY) THE BASE ADDRESS OF COM_MSGBUF!

Ø158 4DD5      LD    SWAP_OUT_MSG.S_OUT_CODE(R13), #SWAP_OUT_CODE
Ø15A ØØØØ
Ø15C ØØ37
Ø15E 3126      LD    R6,R2(#Ø)  !INDEX TO MM_HANDLE ENTRY!
Ø16Ø ØØØØ
Ø162 6FD6      LD    SWAP_OUT_MSG.SO_MM_HANDLE[Ø](R13),R6
Ø164 ØØØ4
Ø166 3126      LD    R6,R2(#2)
Ø168 ØØØ2
Ø16A 6FD6      LD    SWAP_OUT_MSG.SO_MM_HANDLE[1](R13),R6
Ø16C ØØØ6
Ø16E 3126      LD    R6,R2(#4)
Ø17Ø ØØØ4
Ø172 6FD6      LD    SWAP_OUT_MSG.SO_MM_HANDLE[2](R13),R6
Ø174 ØØØ8
Ø176 6FD1      LD    SWAP_OUT_MSG.SO_DBR_NO(R13),R1
Ø178 ØØØ2
Ø17A A1D8      LD    R8,R13
Ø17C 5FØØ      CALL  PERFORM_IPC   !R8: ^COM_MSGBUF!
Ø17E Ø18C'
```

```
                    !RETRIEVE SUCCESS_CODE FROM RETURNED MESSAGE!
0180 8D08           CLR    R0
0182 60D8           LDB    RL0,RET_SUC_CODE.SUC_CODE(R13)
0184 0000
0186 010F           ADD    R15,#SIZEOF COM_MSG !RESTORE STACK STATE!
0188 0010
018A 9E08           RET
018C           END  MM_SWAP_OUT
```

```
018C          PERFORM_IPC                        PROCEDURE
             !*************************************
              * SERVICE ROUTINE TO ARRANGE AND     *
              * PERFORM IPC WITH THE MEM MGR PROC  *
              *************************************
              * REGISTER USE:                      *
              * PARAMETERS                         *
              *   R8: ^COM_MSG(INPUT)              *
              * LOCAL USE                          *
              *   R1,R2: WORK REGS                 *
              *   R4: ^G_AST_LOCK                  *
              *   R13: ^COM_MSGBUF                 *
              *************************************!

              ENTRY
018C 93FD      PUSH   @R15,R13   !^COM_MSGBUF!
018E 5F00      CALL   GET_CPU_NO !RET-R1:CPU_NO!
0190 0000*
0192 A112      LD     R2,R1
0194 6121      LD     R1,MM_CPU_TBL(R2)   !MM_VP_ID!
0196 0000'
0198 7604      LDA    R4,G_AST_LOCK
019A 0000*
019C 5F00      CALL   K_LOCK
019E 0000*
01A0 5F00      CALL   SIGNAL  !R1:MM_VP_ID,R8:^COM_MSGBUF!
01A2 0000*
01A4 97FD      POP    R13,@R15
01A6 A1D8      LD     R8,R13  !^COM_MSGBUF!
01A8 93FD      PUSH   @R15,R13
01AA 5F00      CALL   WAIT  !R8:^COM_MSGBUF!
01AC 0000*
01AE 7604      LDA    R4,G_AST_LOCK
01B0 0000*
01B2 5F00      CALL   K_UNLOCK
01B4 0000*
01B6 97FD      POP    R13,@R15
01B8 9E08      RET
01BA        END PERFORM_IPC
```

```
01BA            MM_ALLOCATE      PROCEDURE
               !*****************************
                * ALLOCATES BLOCKS OF CPU*
                * LOCAL MEMORY.  EACH    *
                * BLOCK CONTAINS 256     *
                * BYTES OF MEMORY.       *
                **************************
                * PARAMETERS:            *
                *   R3: # OF BLOCKS      *
                * RETURNS:               *
                *   R2: STARTING ADDR    *
                * LOCAL:                 *
                *   R4: BLOCK POINTER    *
                **************************!
                ENTRY
                 ! NOTE: THIS PROCEDURE IS ONLY A STUB
                   OF THE ORIGINALLY DESIGNED MEMORY
                   ALLOCATING MECHANISM.  IT IS USED
                   BY THE PROCESS MANAGEMENT DEMONSTRATION
                   TO ALLOCATE CPU LOCAL MEMORY FOR ALL
                   MEMORY ALLOCATION REQUIREMENTS. IN AN
                   ACTUAL SASS ENVIRONMENT, THIS WOULD
                   BE BETTER SERVED TO HAVE SEPARATE
                   ALLOCATION PROCEDURES FOR KERNEL AND
                   SUPERVISOR NEEDS. (E.G., KERNEL_ALLOCATE
                   AND SUPERVISOR_ALLOCATE). !
                 ! COMPUTE SIZE OF MEMORY REQUESTED !
01BA B331        SLL     R3, #BLOCK_SIZE
01BC 0008
                 ! COMPUTE OFFSET OF MEMORY THAT IS
                   TO BE ALLOCATED !
01BE 6104        LD      R4, NEXT_BLOCK  !OFFSET!
01C0 0F00'
01C2 7642        LDA     R2, MEM_POOL(R4) !START ADDR!
01C4 0000'
01C6 8134        ADD     R4, R3  !UPDATE OFFSET!
                 ! UPDATE OFFSET IN SECTION OF AVAILABLE
                   MEMORY TO INDICATE THAT CURRENTLY
                   REQUESTED MEMORY IS NOW ALLOCATED !
01C8 6F04        LD      NEXT_BLOCK, R4  !SAVE OFFSET!
01CA 0F00'
01CC 9E08        RET
01CE            END MM_ALLOCATE
```

```
01CE          MM_TICKET             PROCEDURE
             !***************************
              * RETURNS CURRENT VALUE OF  *
              * SEGMENT SEQUENCER AND     *
              * INCREMENTS SEQUENCER VALUE*
              * FOR NEXT TICKET OPERATION *
              ****************************
              * PARAMETERS:               *
              *  R1: SEG HANDLE PTR       *
              * RETURNS:                  *
              *  RR4: TICKET VALUE        *
              * LOCAL VARIABLES:          *
              *  RR6: SEQUENCER VALUE     *
              *  R8: G_AST ENTRY #        *
              ****************************!
              ENTRY
               ! SAVE HANDLE PTR !
01CE 93F1      PUSH    @R15, R1
               ! LOCK G_AST !
01D0 7604      LDA   R4, G_AST_LOCK
01D2 0000*
01D4 5F00      CALL  K_LOCK
01D6 0000*
               ! RESTORE HANDLE PTR !
01D8 97F1      POP   R1, @R15
               ! GET G_AST ENTRY # !
01DA 6118      LD    R8, MM_HANDLE.ENTRY_NO(R1)
01DC 0004
               ! GET TICKET VALUE !
01DE 5486      LDL   RR6, G_AST.SEQUENCER(R8)
01E0 0014*
               ! SET RETURN REGISTER VALUE !
01E2 9464      LDL   RR4, RR6
               !ADVANCE SEQUENCER FOR NEXT
                TICKET OPERATION!
01E4 1606      ADDL  RR6, #1
01E6 0000
01E8 0001
               ! SAVE NEW SEQUENCER VALUE IN G_AST !
01EA 5D86      LDL   G_AST.SEQUENCER(R8), RR6
01EC 0014*
               ! UNLOCK G_AST !
               ! SAVE RETURN VALUES !
01EE 91F4      PUSHL @R15, RR4
01F0 7604      LDA   R4, G_AST_LOCK
01F2 0000*
01F4 5F00      CALL  K_UNLOCK
01F6 0000*
               ! RETRIEVE RETURN VALUES !
01F8 95F4      POPL  RR4, @R15
01FA 9E08      RET
01FC          END  MM_TICKET
```

```
01FC          MM_READ_EVENTCOUNT    PROCEDURE
             !********************************
              * READS CURRENT VALUE OF THE   *
              * EVENTCOUNT SPECIFIED BY THE  *
              * USER.                        *
              ********************************
              * PARAMETERS:                  *
              *  R1: SEG HANDLE PTR          *
              *  R2: INSTANCE (EVENT #)      *
              ********************************
              * RETURNS:                     *
              *  RR4: EVENTCOUNT VALUE       *
              ********************************
              * LOCAL VARIABLES:             *
              *  RR6: SEQUENCER VALUE        *
              *  R8: G_AST ENTRY #           *
              ********************************!

              ENTRY
               ! SAVE INPUT PARAMETERS !
01FC 93F1      PUSH  @R15, R1
01FE 93F2      PUSH  @R15, R2
               ! LOCK G_AST !
0200 7604      LDA    R4, G_AST_LOCK
0202 0000*
0204 5F00      CALL  K_LOCK
0206 0000*
               ! RESTORE INPUT PARAMETERS !
0208 97F2      POP    R2, @R15
020A 97F1      POP    R1, @R15
               ! GET G_AST ENTRY # !
020C 6118      LD     R8, MM_HANDLE.ENTRY_NO(R1)
020E 0004
               ! READ EVENTCOUNT !
               ! CHECK WHICH EVENT # !
               IF  R2
0210 0B02       CASE  #INSTANCE1 THEN
0212 0001
0214 5E0E
0216 0224'
0218 5484         LDL    RR4, G_AST.EVENT1(R8)
021A 0018*
021C 2100         LD     R0, #SUCCEEDED
021E 0002
0220 5E08       CASE  #INSTANCE2 THEN
0222 023C'
0224 0B02
0226 0002
0228 5E0E
022A 0238'
022C 5484         LDL    RR4, G_AST.EVENT2(R8)
```

```
022E 001C*
0230 2100        LD     R0, #SUCCEEDED
0232 0002
0234 5E08       ELSE  !INVALID INPUT!
0236 023C
0238 2100        LD     R0, #INVALID_INSTANCE
023A 005F
               FI
               ! NOTE: NO VALUE IS RETURNED IF
                 USER SPECIFIED INVALID EVENT #!
               ! SAVE RETURN VALUES !
023C 91F4      PUSHL @R15, RR4
               ! UNLOCK G_AST !
023E 7604      LDA    R4, G_AST_LOCK
0240 0000*
0242 5F00      CALL  K_UNLOCK
0244 0000*
               ! RESTORE RETURN VALUES !
0246 95F4      POPL  RR4, @R15
0248 9E08      RET
024A          END  MM_READ_EVENTCOUNT
```

```
024A           MM_ADVANCE                    PROCEDURE
              !*************************************
               * DETERMINES G_AST OFFSET FROM      *
               * SEGMENT HANDLE AND INCREMENTS     *
               * THE INSTANCE(EVENT #) SPECIFIED   *
               * BY THE CALLER.  THIS IN EFFECT    *
               * ANNOUNCES THE OCCURRENCE OF THE   *
               * EVENT.  THE NEW VALUE OF THE      *
               * EVENTCOUNT IS RETURNED TO THE     *
               * CALLER.                           *
               *************************************
               * PARAMETERS:                       *
               *  R1: HANDLE POINTER               *
               *  R2: INSTANCE (EVENT #)           *
               *************************************
               * RETURNS:                          *
               *  RR2: NEW EVENTCOUNT VALUE        *
               *************************************!

               ENTRY
                ! SAVE INPUT PARAMETERS !
024A 93F1       PUSH  @R15, R1
024C 93F2       PUSH  @R15, R2
                ! LOCK G_AST !
024E 7604       LDA   R4, G_AST_LOCK
0250 0000*
0252 5F00       CALL  K_LOCK
0254 0000*
                ! RESTORE INPUT PARAMETERS !
0256 97F2       POP   R2, @R15
0258 97F1       POP   R1, @R15
                ! GET G_AST OFFSET !
025A 6114       LD    R4, MM_HANDLE.ENTRY_NO(R1)
025C 0004
                ! DETERMINE INSTANCE !
                IF  R2
025E 0B02        CASE  #INSTANCE1  THEN
0260 0001
0262 5E0E
0264 027C'
0266 5442         LDL   RR2, G_AST.EVENT1(R4)
0268 0018*
026A 1602         ADDL  RR2, #1
026C 0000
026E 0001
                  ! SAVE NEW EVENTCOUNT !
0270 5D42         LDL   G_AST.EVENT1(R4), RR2
0272 0018*
0274 2100         LD    R0, #SUCCEEDED
0276 0002
0278 5E08        CASE  #INSTANCE2  THEN
```

```
027A 029E'
027C 0B02
027E 0002
0280 5E0E
0282 029A'
0284 5442        LDL   RR2, G_AST.EVENT2(R4)
0286 001C*
0288 1602        ADDL  RR2, #1
028A 0000
028C 0001
                 ! SAVE NEW EVENTCOUNT !
028E 5D42        LDL   G_AST.EVENT2(R4), RR2
0290 001C*
0292 2100        LD    R0, #SUCCEEDED
0294 0002
0296 5E08       ELSE  !INVALID INPUT!
0298 029E'
029A 2100        LD    R0, #INVALID_INSTANCE
029C 005F
               FI
               ! NOTE: AN INVALID INSTANCE VALUE
                 WILL NOT AFFECT EVENT DATA !
               ! UNLOCK G_AST !
029E 7604      LDA   R4, G_AST_LOCK
02A0 0000*
02A2 5F00      CALL  K_UNLOCK
02A4 0000*
02A6 9E08      RET
02A8          END MM_ADVANCE
             END DIST_MM
```

APPENDIX D - GATE KEEPER LISTINGS

```
Z8000ASM  2.02
LOC     OBJ CODE    STMT SOURCE STATEMENT

         KERNEL_GATE_KEEPER       MODULE

         $LISTON $TTY

         CONSTANT
           ADVANCE_CALL           := 1
           AWAIT_CALL             := 2
           CREATE_SEG_CALL        := 3
           DELETE_SEG_CALL        := 4
           MAKE_KNOWN_CALL        := 5
           READ_CALL              := 6
           SM_SWAP_IN_CALL        := 7
           SM_SWAP_OUT_CALL       := 8
           TERMINATE_CALL         := 9
           TICKET_CALL            := 10
           WRITE_CALL             := 11
           WRITELN_CALL           := 12
           CRLF_CALL              := 13
           WRITE                  := %0FC8 !PRINT CHAR!
           WRITELN                := %0FC0 !PRINT MSG!
           CRLF                   := %0FD4 !CAR RET/LINE FEED!
           MONITOR                := %A902
           REGISTER_BLOCK         := 32
           TRAP_CODE_OFFSET       := 36
           INVALID_KERNEL_ENTRY   := %BAD

         GLOBAL
           GATE_KEEPER_ENTRY      LABEL

         EXTERNAL
           ADVANCE                PROCEDURE
           AWAIT                  PROCEDURE
           CREATE_SEG             PROCEDURE
           DELETE_SEG             PROCEDURE
           MAKE_KNOWN             PROCEDURE
           READ                   PROCEDURE
           SM_SWAP_IN             PROCEDURE
           SM_SWAP_OUT            PROCEDURE
           TERMINATE              PROCEDURE
           TICKET                 PROCEDURE
           KERNEL_EXIT            LABEL

         INTERNAL
         $SECTION KERNEL_GATE_PROC
```

```
0000          GATE_KEEPER_MAIN          PROCEDURE

              ENTRY
              GATE_KEEPER_ENTRY:
                ! SAVE REGISTERS !
0000 030F       SUB   R15, #REGISTER_BLOCK
0002 0020
0004 1CF9       LDM   @R15, R1, #16
0006 010F
                ! SAVE NSP !
0008 93F2       PUSH  @R15, R2
000A 7D27       LDCTL R2, NSP
                ! RESTORE INPUT REGISTERS !
000C 2DF2       EX    R2, @R15
                ! SAVE REGISTER 2 !
000E 93F2       PUSH  @R15, R2
                ! GET SYSTEM TRAP CODE !
0010 31F2       LD    R2, R15(#TRAP_CODE_OFFSET)
0012 0024
                ! REMOVE SYSTEM CALL IDENTIFIER FROM
                  SYSTEM TRAP INSTRUCTION !
0014 8C28       CLRB  RH2
                ! NOTE: THIS LEAVES THE USER VISIBLE
                  EXTENDED INSTRUCTION NUMBER IN R2 !
                ! DECODE AND EXECUTE EXTENDED INSTRUCTION !
                IF  R2
                ! NOTE: THE INITIAL VALUE FOR REGISTER 2
                  WILL BE RESTORED WHEN THE APPROPRIATE
                  CONDITION IS FOUND !
0016 0B02        CASE  #ADVANCE_CALL  THEN
0018 0001
001A 5E0E
001C 0028'
001E 97F2          POP   R2, @R15
0020 5F00          CALL  ADVANCE
0022 0000*
0024 5E08        CASE  #AWAIT_CALL  THEN
0026 010C'
0028 0B02
002A 0002
002C 5E0E
002E 003A'
0030 97F2          POP   R2, @R15
0032 5F00          CALL  AWAIT
0034 0000*
0036 5E08        CASE  #CREATE_SEG_CALL  THEN
0038 010C'
003A 0B02
003C 0003
003E 5E0E
0040 004C'
```

```
0042 97F2        POP   R2, @R15
0044 5F00        CALL  CREATE_SEG
0046 0000*
0048 5E08     CASE  #DELETE_SEG_CALL  THEN
004A 010C'
004C 0B02
004E 0004
0050 5E0E
0052 005E'
0054 97F2        POP   R2, @R15
0056 5F00        CALL  DELETE_SEG
0058 0000*
005A 5E08     CASE  #MAKE_KNOWN_CALL  THEN
005C 010C'
005E 0B02
0060 0005
0062 5E0E
0064 0070'
0066 97F2        POP   R2, @R15
0068 5F00        CALL  MAKE_KNOWN
006A 0000*
006C 5E08     CASE  #READ_CALL  THEN
006E 010C'
0070 0B02
0072 0006
0074 5E0E
0076 0082'
0078 97F2        POP   R2, @R15
007A 5F00        CALL  READ
007C 0000*
007E 5E08     CASE  #SM_SWAP_IN_CALL  THEN
0080 010C'
0082 0B02
0084 0007
0086 5E0E
0088 0094'
008A 97F2        POP   R2, @R15
008C 5F00        CALL  SM_SWAP_IN
008E 0000*
0090 5E08     CASE  #SM_SWAP_OUT_CALL  THEN
0092 010C'
0094 0B02
0096 0008
0098 5E0E
009A 00A6'
009C 97F2        POP   R2, @R15
009E 5F00        CALL  SM_SWAP_OUT
00A0 0000*
00A2 5E08     CASE  #TERMINATE_CALL  THEN
00A4 010C'
00A6 0B02
```

```
00A8 0009
00AA 5E0E
00AC 00B8'
00AE 97F2       POP   R2, @R15
00B0 5F00       CALL  TERMINATE
00B2 0000*
00B4 5E08     CASE  #TICKET_CALL  THEN
00B6 010C'
00B8 0B02
00BA 000A
00BC 5E0E
00BE 00CA'
00C0 97F2       POP   R2, @R15
00C2 5F00       CALL  TICKET
00C4 0000*
00C6 5E08     CASE  #WRITE_CALL   THEN
00C8 010C'
00CA 0B02
00CC 000B
00CE 5E0E
00D0 00DC'
00D2 97F2       POP   R2, @R15
00D4 5F00       CALL  WRITE
00D6 0FC8
00D8 5E08     CASE  #WRITELN_CALL  THEN
00DA 010C'
00DC 0B02
00DE 000C
00E0 5E0E
00E2 00EE'
00E4 97F2       POP   R2, @R15
00E6 5F00       CALL  WRITELN
00E8 0FC0
00EA 5E08     CASE  #CRLF_CALL   THEN
00EC 010C'
00EE 0B02
00F0 000D
00F2 5E0E
00F4 0100'
00F6 97F2       POP   R2, @R15
00F8 5F00       CALL  CRLF
00FA 0FD4
00FC 5E08     ELSE !INVALID KERNEL INVOCATION!
00FE 010C'
                ! RETURN TO MONITOR !
                ! NOTE: THIS RETURN TO MONITOR IS
                  FOR STUB USE ONLY.  AN INVALID
                  KERNEL INVOCATION WOULD NORMALLY
                  RETURN TO USER. !
0100 7601       LDA   R1, $
0102 0100'
```

```
0104 2100        LD    R0, #INVALID_KERNEL_ENTRY
0106 0BAD
0108 5F00        CALL  MONITOR
010A A902
              FI
              ! SAVE REGISTERS ON KERNEL STACK !
              ! SAVE R1 !
010C 93F1     PUSH  @R15, R1
              ! GET ADDRESS OF REGISTER BLOCK !
010E 34F1     LDA   R1, R15(#4)
0110 0004
              ! SAVE REGISTERS IN REGISTER BLOCK
                ON KERNEL STACK. !
0112 1C19     LDM   @R1, R1, #16
0114 010F
              ! RESTORE R1 BUT MAINTAIN ADDRESS
                OF REGISTER BLOCK !
0116 2DF1     EX     R1, @R15
              ! SAVE R1 ON STACK !
0118 33F1     LD     R15(#4), R1
011A 0004
              ! RESTORE REGISTER BLOCK ADDRESS !
011C 97F1     POP    R1, @R15
              ! SAVE VALID EXIT SP VALUE !
011E 33F1     LD     R15(#30), R1
0120 001E
              ! EXIT KERNEL BY MEANS OF HARDWARE
                PREEMPT HANDLER !
0122 5E08     JP     KERNEL_EXIT
0124 0000*
0126        END  GATE_KEEPER_MAIN
           END  KERNEL_GATE_KEEPER
```

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

        USER_GATE       MODULE

        $LISTON  $TTY

        CONSTANT
          ADVANCE_CALL            := 1
          AWAIT_CALL              := 2
          CREATE_SEG_CALL         := 3
          DELETE_SEG_CALL         := 4
          MAKE_KNOWN_CALL         := 5
          READ_CALL               := 6
          SM_SWAP_IN_CALL         := 7
          SM_SWAP_OUT_CALL        := 8
          TERMINATE_CALL          := 9
          TICKET_CALL             := 10
          WRITE_CALL              := 11
          WRITELN_CALL            := 12
          CRLF_CALL               := 13

        GLOBAL
        $SECTION USER_GATE_PROC

0000        ADVANCE          PROCEDURE
           !************************
            * PARAMETERS:           *
            *  R1:SEGMENT #         *
            *  R2:INSTANCE (ENTRY#)*
            ************************
            * RETURNS:              *
            *  R0:SUCCESS CODE      *
            ************************!
            ENTRY
0000 7F01     SC     #ADVANCE_CALL
0002 9E08     RET
0004        END  ADVANCE

0004        AWAIT            PROCEDURE
           !************************
            * PARAMETERS:           *
            *  R1:SEGMENT #         *
            *  R2:INSTANCE          *
            *  RR4:SPECIFIED VALUE *
            ************************
            * RETURNS:              *
            *  R0:SUCCESS CODE      *
            ************************!
            ENTRY
```

```
0004 7F02     SC    #AWAIT_CALL
0006 9E08     RET
0008         END  AWAIT

0008         CREATE_SEG      PROCEDURE
            !*************************
             * PARAMETERS:            *
             *  R1:MENTOR_SEG_NO      *
             *  R2:ENTRY_NO           *
             *  R3:SIZE               *
             *  RR4:CLASS             *
             **************************
             * RETURNS:               *
             *  R0:SUCCESS CODE       *
             *************************!
             ENTRY
0008 7F03     SC    #CREATE_SEG_CALL
000A 9E08     RET
000C         END  CREATE_SEG

000C         DELETE_SEG      PROCEDURE
            !*************************
             * PARAMETERS:            *
             *  R1:MENTOR_SEG_NO      *
             *  R2:ENTRY_NO           *
             **************************
             * RETURNS:               *
             *  R0:SUCCESS CODE       *
             *************************!
             ENTRY
000C 7F04     SC    #DELETE_SEG_CALL
000E 9E08     RET
0010         END  DELETE_SEG

0010         MAKE_KNOWN      PROCEDURE
            !*************************
             * PARAMETERS:            *
             *  R1:MENTOR_SEG_NO      *
             *  R2:ENTRY_NO           *
             *  R3:ACCESS DESIRED     *
             **************************
             * RETURNS:               *
             *  R0:SUCCESS CODE       *
             *  R1:SEGMENT #          *
             *  R2:ACCESS ALLOWED     *
             *************************!
             ENTRY
0010 7F05     SC    #MAKE_KNOWN_CALL
0012 9E08     RET
0014         END  MAKE_KNOWN
```

```
0014         READ              PROCEDURE
            !************************
             * PARAMETERS:           *
             *   R1:SEGMENT #        *
             *   R2:INSTANCE         *
             *************************
             * RETURNS:              *
             *   R0:SUCCESS CODE     *
             *   RR4:EVENTCOUNT      *
             ************************!
             ENTRY
0014 7F06      SC     #READ_CALL
0016 9E08      RET
0018         END  READ

0018         SM_SWAP_IN        PROCEDURE
            !************************
             * PARAMETERS:           *
             *   R1:SEGMENT #        *
             *************************
             * RETURNS:              *
             *   R0:SUCCESS CODE     *
             ************************!
             ENTRY
0018 7F07      SC     #SM_SWAP_IN_CALL
001A 9E08      RET
001C         END  SM_SWAP_IN

001C         SM_SWAP_OUT       PROCEDURE
            !************************
             * PARAMETERS:           *
             *   R1:SEGMENT #        *
             *************************
             * RETURNS:              *
             *   R0:SUCCESS CODE     *
             ************************!
             ENTRY
001C 7F08      SC     #SM_SWAP_OUT_CALL
001E 9E08      RET
0020         END  SM_SWAP_OUT

0020         TERMINATE         PROCEDURE
            !************************
             * PARAMETERS:           *
             *   R1:SEGMENT #        *
             *************************
             * RETURNS:              *
             *   R0:SUCCESS CODE     *
             ************************!
             ENTRY
0020 7F09      SC     #TERMINATE_CALL
```

```
0022 9E08     RET
0024         END  TERMINATE

0024         TICKET          PROCEDURE
            !*************************
             * PARAMETERS:           *
             *  R1:SEGMENT #         *
             *************************
             * RETURNS:              *
             *  R0:SUCCESS CODE      *
             *  RR4:TICKET VALUE     *
             *************************!
             ENTRY
0024 7F0A      SC     #TICKET_CALL
0026 9E08      RET
0028         END  TICKET

0028         WRITE           PROCEDURE
             ENTRY
0028 7F0B      SC     #WRITE_CALL
002A 9E08      RET
002C         END  WRITE

002C         WRITELN         PROCEDURE
             ENTRY
002C 7F0C      SC     #WRITELN_CALL
002E 9E08      RET
0030         END  WRITELN

0030         CRLF            PROCEDURE
             ENTRY
0030 7F0D      SC     #CRLF_CALL
0032 9E08      RET
0034         END  CRLF
```

APPENDIX E - BOOTSTRAP_LOADER LISTINGS

```
Z8000ASM  2.02
LOC     OBJ CODE     STMT SOURCE STATEMENT

                          BOOTSTRAP_LOADER   MODULE

              $LISTON $TTY
              CONSTANT

                         ! ******** SYSTEM PARAMETERS ******** !
                         NR_CPU            := 2
                         NR_VP             := NR_CPU*4
                         NR_AVAIL_VP       := NR_CPU*2
                         MAX_DBR_NR        := 10
                         STACK_SEG         := 1
                         STACK_SEG_SIZE    := %100
                         STACK_BLOCK       := STACK_SEG_SIZE/256

                           ! * * OFFSETS IN STACK SEG * * !
                         STACK_BASE        := STACK_SEG_SIZE-%10
                         STATUS_REG_BLOCK:= STACK_SEG_SIZE-%10
                         INTERRUPT_FRAME   := STACK_BASE-4
                         INTERRUPT_REG     := INTERRUPT_FRAME-34
                         N_S_P             := INTERRUPT_REG-2
                         F_C_W             := STACK_SEG_SIZE-%E

                        ! ****** SYSTEM CONSTANTS ****** !
                         ON                := %FFFF
                         OFF               := 0
                         READY             := 1
                         NIL               := %FFFF
                         INVALID           := %EEEE
                         KERNEL_FCW        := %5000
                         AVAILABLE         := 0
                         ALLOCATED         := %FF
                         SC_OFFSET         := 12

                TYPE

                         MESSAGE  ARRAY [16   BYTE]
                         ADDRESS  WORD
                         MM_VP_ID WORD
                         VP_INDEX          INTEGER
                         MSG_INDEX         INTEGER
```

```
    MSG_TABLE RECORD
     [ MSG            MESSAGE
       SENDER         VP_INDEX
       NEXT_MSG       MSG_INDEX
       FILLER         ARRAY [6, WORD]
     ]

VP_TABLE RECORD
    [ DBR    ADDRESS
      PRI             WORD
      STATE           WORD
      IDLE_FLAG       WORD
      PREEMPT         WORD
      PHYS_PROCESSOR  WORD
      NEXT_READY_VP VP_INDEX
      MSG_LIST        MSG_INDEX
      EXT_ID          WORD
      FILLER_1        ARRAY [7, WORD]
    ]

EXTERNAL
    GET_DBR_ADDR      PROCEDURE
    CREATE_STACK      PROCEDURE
    LIST_INSERT       PROCEDURE
    ALLOCATE_MMU      PROCEDURE
    UPDATE_MMU_IMAGE PROCEDURE
    MM_ALLOCATE       PROCEDURE
    MM_ENTRY          LABEL
    IDLE_ENTRY        LABEL
    PREEMPT_RET       LABEL
    BOOTSTRAP_ENTRY LABEL
    GATE_KEEPER_ENTRY LABEL
    NEXT_BLOCK        WORD
    MM_CPU_TBL ARRAY[NR_CPU MM_VP_ID]


   VPT        RECORD
    [ LOCK           WORD
      RUNNING_LIST ARRAY[NR_CPU WORD]
      READY_LIST   ARRAY[NR_CPU WORD]
      FREE_LIST    MSG_INDEX
      VIRT_INT_VEC ARRAY[1, ADDRESS]
      FILLER_2       WORD
      VP         ARRAY [NR_VP, VP_TABLE]
      MSG_Q          ARRAY [NR_VP, MSG_TABLE]
    ]
```

```
                EXT_VP_LIST     ARRAY[NR_AVAIL_VP WORD]
                NEXT_AVAIL_MMU  ARRAY[MAX_DBR_NR  BYTE]

                PRDS     RECORD
                   [PHYS_CPU_ID WORD
                    LOG_CPU_ID  INTEGER
                    VP_NR       WORD
                    IDLE_VP     VP_INDEX]


            INTERNAL
            $SECTION LOADER_DATA

           ! NOTE: THESE DECLARATIONS WILL NOT WORK
             IN A TRUE MULTIPROCESSOR ENVIRONMENT AS
             THEY ARE SUBJECT TO A "CALL."  THEY MUST
             BE DECLARED AS A SHARED GLOBAL DATABASE
             WITH "RACE" PROTECTION (E.G., LOCK). !

0000       NEXT_AVAIL_VP   INTEGER
0002       NEXT_EXT_VP     INTEGER
```

```
        $SECTION  LOADER_INT
          INTERNAL
0000                BOOTSTRAP                    PROCEDURE
                   !*******************************
                    * CREATES KERNEL PROCESSES AND *
                    * INITIALIZES KERNEL DATABASES.*
                    * INCLUDES INITIALIZATION OF   *
                    * VIRTUAL PROCESSOR TABLE,     *
                    * EXTERNAL VP LIST, AND MMU    *
                    * IMAGES.  ALLOCATES MMU IMAGE *
                    * AND CREATES KERNEL DOMAIN    *
                    * STACK FOR KERNEL PROCESSES.  *
                    *******************************!

                    ENTRY
                    ! INITIALIZE PRDS AND MMU POINTER !
                    ! NOTE: THE FOLLOWING CONSTANTS ARE
                      ONLY TO BE INITIALIZED ONCE.  THIS
                      WILL OCCUR DURING SYSTEM INITIALIZATION!
0000 4D05           LD         PRDS.PHYS_CPU_ID, #%FFFF
0002 0000*
0004 FFFF
                     ! NOTE: LOGICAL CPU NUMBERS ARE ASSIGNED
                       IN INCREMENTS OF 2 TO FACILITATE INDEXING
                       (OFFSETS) INTO LISTS SUBSCRIPTED BY
                       LOGICAL CPU NUMBER. !
0006 4D05           LD         PRDS.LOG_CPU_ID, #2
0008 0002*
000A 0002
                     ! SPECIFY NUMBER OF VIRTUAL PROCESSORS
                       ASSOCIATED WITH PHYSICAL CPU. !
000C 4D05           LD         PRDS.VP_NR, #2
000E 0004*
0010 0002
0012 4D08           CLR        NEXT_BLOCK
0014 0000*
0016 4D08           CLR        NEXT_AVAIL_VP
0018 0000'
001A 4D08           CLR        NEXT_EXT_VP
001C 0002'
                    ! ESTABLISH GATE KEEPER AS SYSTEM CALL
                      TRAP HANDLER !
                    ! GET BASE OF PROGRAM STATUS AREA !
001E 7D15           LDCTL      R1, PSAP

                    ! ADD SYSTEM CALL OFFSET TO PSA BASE ADDR !
0020 0101           ADD        R1, #SC_OFFSET
0022 000C
                    ! STORE KERNEL FCW IN PSA !
0024 0D15           LD         @R1, #KERNEL_FCW
0026 5000
```

```
                  ! STORE ADDRESS OF GATE KEEPER IN PROGRAM
                    STATUS AREA AS SYSTEM TRAP HANDLER !
0028 A911         INC        R1, #2
002A 0D15         LD         @R1, #GATE_KEEPER_ENTRY
002C 0000*
002E 8D18         CLR        R1   ! NEXT_AVAIL_MMU INDEX !

                  ! INITIALIZE ALL MMU IMAGES AS AVAILABLE !
               SET_MMU_MAP:

                   DO
0030 4C15           LDB      NEXT_AVAIL_MMU(R1), #AVAILABLE
0032 0000*
0034 0000
0036 A910           INC      R1, #1
                    ! CHECK FOR END OF TABLE !
0038 0B01           CP       R1, #MAX_DBR_NR
003A 000A
003C 5E0E           IF EQ THEN EXIT FROM SET_MMU_MAP  FI
003E 0044'
0040 5E08
0042 0046'
0044 E8F5         OD

                  ! CREATE MEMORY MANAGER PROCESS !
0046 2103         LD         R3, #STACK_BLOCK
0048 0001
                  ! ALLOCATE AND INITIALIZE KERNEL
                    DOMAIN STACK SEGMENT !
004A 5F00         CALL       MM_ALLOCATE  !R3: # OF BLOCKS
004C 0000*
                                                RETURNS
                                                R2: START ADDR!
004E A121         LD         R1, R2
0050 2103         LD         R3, #KERNEL_FCW
0052 5000
0054 7604         LDA        R4, MM_ENTRY
0056 0000*
0058 6105         LD         R5, %FFFF  !NSP!
005A FFFF
005C 7606         LDA        R6, PREEMPT_RET
005E 0000*
0060 93F1         PUSH       @R15, R1 !SAVE STACK ADDR!
0062 030F         SUB        R15, #8
0064 0008
0066 1CF9         LDM        @R15, R3, #4
0068 0303
006A A1F0         LD         R0, R15

                  ! NOTE: ARGLIST FOR CREATE_STACK INCLUDES
                    KERNEL_FCW, INITIAL IC, NSP, AND INITIAL
                  RETURN POINT. !
006C 5F00      CALL       CREATE_STACK   ! (R0: ARGUMENT PTR
006E 0000*
                                            R1: TOP OF STACK
                                            R2-R14: INITIAL
                                            REG.STATES !
0070 010F      ADD        R15, #8  !OVERLAY ARGUMENTS!
0072 0008
               ! ALLOCATE MMU_IMAGE !
0074 5F00      CALL       ALLOCATE_MMU    !RETURNS:
0076 0000*
                                           (R0: DBR #) !
0078 2101      LD         R1, #STACK_SEG    ! SEGMENT NO. !
007A 0001
007C 97F2      POP        R2, @R15  !GET STACK ADDR!
007E 2103      LD         R3, #0    ! WRITE ATTRIBUTE !
0080 0000
               ! SPECIFY NUMBER OF BLOCKS. COUNT STARTS
                 FROM ZERO. (I.E.,1 BLOCK=0, 2=1, ETC.)!
0082 2104      LD         R4, #STACK_BLOCK-1
0084 0000
               ! SAVE DBR # !
0086 93F0      PUSH       @R15, R0

               ! CREATE MMU ENTRY FOR MM STACK SEGMENT !
0088 5F00      CALL       UPDATE_MMU_IMAGE  !(R0: DBR #
008A 0000*
                                            R1: SEGMENT #
                                            R2: SEG ADDRESS
                                            R3: SEG ATTRIBUTES
                                            R4: SEG LIMITS)  !
               ! RESTORE DBR # !
008C 97F0      POP         R0, @R15

               ! GET ADDRESS OF MMU IMAGE !
008E 5F00      CALL       GET_DBR_ADDR  ! (R0: DBR #)
0090 0000*
                                           RETURNS:
                                           (R1: DBR ADDRESS) !
               ! PREPARE VP TABLE ENTRIES FOR MM !
0092 2102      LD         R2, #2      ! PRIORITY !
0094 0002
0096 2105      LD         R5, #OFF    ! PREEMPT !
0098 0000
009A 2106      LD         R6, #OFF  ! KERNEL PROCESS !
009C 0000

               ! UPDATE VPT !
009E 5F00      CALL       UPDATE_VP_TABLE  !(R1: DBR
00A0 01CA'
                                            R2: PRIORITY
```

```
                                                     R5: PREEMPT FLAG
                                                     R6: EXT_VP FLAG)
                                                     RETURNS:
                                                     R9: VP_ID !
                ! INITIALIZE MM_CPU_TBL IN DISTRIBUTED MEMORY
                  MANAGER WITH VP ID OF MM PROCESS !
                ! GET LOGICAL CPU # !
00A2 610A       LD R10, PRDS.LOG_CPU_ID
00A4 0002*
00A6 6FA9       LD          MM_CPU_TBL(R10), R9
00A8 0000*
                ! CREATE IDLE PROCESS !
00AA 2103       LD          R3, #STACK_BLOCK
00AC 0001
00AE 5F00       CALL        MM_ALLOCATE !R3: # OF BLOCKS
00B0 0000*
                                              RETURNS
                                              R2: START ADDR!
00B2 A121       LD          R1, R2
00B4 2103       LD          R3, #KERNEL_FCW
00B6 5000
00B8 7604       LDA         R4, IDLE_ENTRY
00BA 0000*
00BC 2105       LD          R5, #%FFFF !NSP!
00BE FFFF
00C0 7606       LDA         R6, PREEMPT_RET
00C2 0000*
00C4 93F1       PUSH        @R15, R1 !SAVE STACK ADDR!
00C6 030F       SUB         R15, #8
00C8 0008
00CA 1CF9       LDM         @R15, R3, #4
00CC 0303
00CE A1F0       LD          R0, R15

                ! INITIALIZE IDLE STACK VALUES !
00D0 5F00       CALL        CREATE_STACK  ! (R0: ARGUMENT PTR
00D2 0000*
                                                   R1: TOP OF STACK
                                                   R2-R14: INITIAL
                                                   REG. STATES !
00D4 010F       ADD         R15, #8 !OVERLAY ARGUMENTS!
00D6 0008

                ! ALLOCATE MMU IMAGE FOR IDLE PROCESS !
00D8 5F00       CALL        ALLOCATE_MMU  ! RETURNS R0:DBR # !
00DA 0000*

                ! PREPARE IDLE PROCESS MMU ENTRIES !
00DC 2101       LD          R1, #STACK_SEG  ! SEG # !
00DE 0001
00E0 97F2       POP         R2, @R15  !GET STACK ADDR!
```

```
00E2 2103     LD        R3, #0              ! WRITE ATTRIBUTE !
00E4 0000
00E6 2104     LD        R4, #STACK_BLOCK-1  ! BLOCK LIMITS !
00E8 0000
              ! SAVE DBR # !
00EA 93F0     PUSH      @R15, R0

              ! CREATE MMU IMAGE ENTRY !
00EC 5F00     CALL      UPDATE_MMU_IMAGE  !(R1: SEGMENT #
00EE 0000*
                                            R2: SEG ADDRESS
                                            R3: SEG ATTRIBUTES
                                            R4: SEG LIMITS ) !
              ! RESTORE DBR # !
00F0 97F0     POP       R0, @R15

              ! GET MMU ADDRESS !
00F2 5F00     CALL      GET_DBR_ADDR  ! (R0: DBR #)
00F4 0000*
                                           RETURNS
                                           (R1: DBR ADDRESS) !
              ! PREPARE VPT ENTRIES FOR IDLE PROCESS !
00F6 2102     LD        R2, #0              ! PRIORITY !
00F8 0000
00FA 2105     LD        R5, #OFF            ! PREEMPT !
00FC 0000
00FE 2106     LD        R6, #OFF            ! KERNEL PROC !
0100 0000

              ! CREATE VPT ENTRIES !
0102 5F00     CALL      UPDATE_VP_TABLE  !(R1: DBR
0104 01CA'
                                            R2: PRIORITY
                                            R4: IDLE_FLAG
                                            R5: PREEMPT
                                            R6: EXT_VP FLAG)
                                            RETURNS:
                                            R9: VP_ID !
              ! ENTER VP ID OF IDLE PROCESS IN PRDS !
0106 6F09     LD        PRDS.IDLE_VP, R9
0108 0006*
              ! INITIALIZE IDLE VP'S !
010A 2102     LD        R2, #1              ! PRIORITY !
010C 0001
010E 2105     LD        R5, #ON             ! PREEMPT !
0110 FFFF
0112 2106     LD        R6, #ON             !NON-KERNEL PROC!
0114 FFFF
0116 6100     LD        R0, PRDS.VP_NR
0118 0004*
              ! INITIALIZE VP VALUES !
```

```
                DO
011A 5F00       CALL        UPDATE_VP_TABLE  !(R1: DBR
011C 01CA'
                                                R2: PRIORITY
                                                R4: IDLE_FLAG
                                                R5: PREEMPT
                                                R6: EXT_VP FLAG)
                                                RETURNS:
                                                R9: VP_ID !
011E AB00       DEC         R0, #1
0120 0B00       CP          R0, #0
0122 0000
0124 5E0E       IF EQ !ALL VP'S INITIALIZED! THEN
0126 012C'
0128 5E08          EXIT
012A 012E'
                FI
012C E8F6      OD

               ! INITILIZE VPT HEADER !
               ! GET LOGICAL CPU NUMBER !
012E 6102      LD           R2, PRDS.LOG_CPU_ID
0130 0002*
0132 4D05      LD           VPT.LOCK, #OFF
0134 0000*
0136 0000
0138 4D25      LD           VPT.RUNNING_LIST(R2), #NIL
013A 0002*
013C FFFF
013E 4D25      LD           VPT.READY_LIST(R2), #NIL
0140 0006*
0142 FFFF
0144 4D08      CLR          VPT.FREE_LIST  !HEAD OF MSG LIST!
0146 000A*

             !THREAD VP'S BY PRIORITY AND SET STATES TO READY !
0148 8D28       CLR         R2  !START WITH VP #1!

              THREAD:
                DO
014A 610D         LD        R13, PRDS.LOG_CPU_ID
014C 0002*
014E 76D3         LDA       R3,VPT.READY_LIST(R13)
0150 0006*
0152 7604         LDA       R4,VPT.VP.NEXT_READY_VP
0154 001C*
0156 7605         LDA       R5,VPT.VP.PRI
0158 0012*
015A 7606         LDA       R6,VPT.VP.STATE
015C 0014*
015E 2107         LD        R7,#READY
```

```
0160 0001
                 ! SAVE OBJ ID !
0162 93F2        PUSH    @R15, R2
0164 5F00        CALL    LIST_INSERT !R2: OBJ ID
0166 0000*
                                      R3: LIST_HEAD_PTR ADDR
                                      R4: NEXT_OBJ PTR
                                      R5: PRIORITY_PTR
                                      R6: STATE_PTR
                                      R7: STATE   !
                 ! RESTORE OBJ ID !
0168 97F2        POP     R2, @R15
016A 0102        ADD     R2, #SIZEOF VP_TABLE
016C 0020
016E 0B02        CP      R2, #(NR_VP * (SIZEOF VP_TABLE))
0170 0100
0172 5E0E        IF EQ THEN EXIT FROM THREAD FI
0174 017A'
0176 5E08
0178 017C'
017A E8E7     OD

              ! INITIALIZE VP MESSAGE LIST !
              ! NOTE: ONLY THE THREAD FOR THE MESSAGE
                LIST NEED BE CREATED AS ALL MESSAGES
                ARE INITIALLY AVAILABLE FOR USE. THE
                INITIAL MESSAGE VALUES WERE CREATED
                FOR CLARITY ONLY TO SHOW THAT THE
                MESSAGES HAVE NO USABLE INITIAL VALUE!
017C 8D18     CLR        R1

           MSG_LST_INIT:
              ! NOTE: R1 REPRESENTS CURRENT ENTRY IN
                MSG_LIST, R2 REPRESENTS CURRENT POSITION
                IN MSG_LIST ENTRY, AND R3 REPRESENTS
                NEXT ENTRY IN MSG_LIST. !
               DO
017E A112       LD       R2, R1
0180 A123       LD       R3, R2
0182 0103       ADD      R3, #SIZEOF MESSAGE
0184 0010
               FILL_MSG:
                 DO
0186 4D25        LD      VPT.MSG_Q.MSG(R2), #INVALID
0188 0110*
018A EEEE
018C A921        INC     R2, #2
018E 8B32        CP      R2, R3
0190 5E0E        IF EQ THEN EXIT FROM FILL_MSG FI
0192 0198'
0194 5E08
```

```
0196 019A'
0198 E8F6      OD
019A 4D15      LD          VPT.MSG_Q.SENDER(R1), #NIL
019C 0120*
019E FFFF
01A0 A112      LD          R2, R1
01A2 0101      ADD         R1, #SIZEOF MSG_TABLE
01A4 0020
01A6 0B01      CP          R1, #SIZEOF MSG_TABLE*NR_VP
01A8 0100
                IF EC
01AA 5E0E       THEN
01AC 01BC'
01AE 4D25        LD        VPT.MSG_Q.NEXT_MSG(R2), #NIL
01B0 0122*
01B2 FFFF
01B4 5E08        EXIT FROM MSG_LST_INIT
01B6 01C2'
01B8 5E08       ELSE
01BA 01C0'
01BC 6F21        LD        VPT.MSG_Q.NEXT_MSG(R2), R1
01BE 0122*
                 FI
01C0 E8DE     OD

              ! GET LOGICAL CPU # FOR USE
                BY ITC GETWORK. !
01C2 610D     LD             R13, PRDS.LOG_CPU_ID
01C4 0002*
              ! BOOTSTRAP COMPLETE !
              ! START SYSTEM EXECUTION AT PREEMPT ENTRY !
              ! POINT IN ITC GETWORK PROCEDURE !
01C6 5E08     JP           BOOTSTRAP_ENTRY
01C8 0000*
01CA         END BOOTSTRAP
```

```
01CA           UPDATE VP TABLE         PROCEDURE
              !*********************************
               * INITIALIZES VPT ENTRIES       *
               *********************************
               * REGISTER USE:                 *
               *  PARAMETERS:                  *
               *   R1: DBR ADDRESS             *
               *   R2: PRIORITY                *
               *   R5: PREEMPT FLAG            *
               *   R6: EXTERNAL VP FLAG        *
               *  RETURNS:                     *
               *   R9: ASSIGNED VP ID          *
               *  LOCAL VARIABLES:             *
               *   R7: LOGICAL CPU #           *
               *   R8: EXT_VP_LIST OFFSET      *
               *   R9: VPT OFFSET              *
               *********************************!

               ENTRY
               ! GET OFFSET IN VPT FOR NEXT ENTRY !
01CA 6109      LD         R9, NEXT_AVAIL_VP
01CC 0000'
01CE 6F91      LD         VPT.VP.DBR(R9), R1
01D0 0010*
01D2 6F92      LD         VPT.VP.PRI(R9), R2
01D4 0012*
01D6 6F96      LD         VPT.VP.IDLE_FLAG(R9), R6
01D8 0016*
01DA 6F95      LD         VPT.VP.PREEMPT(R9), R5
01DC 0018*
01DE 6107      LD         R7, PRDS.LOG_CPU_ID
01E0 0002*
01E2 6F97      LD         VPT.VP.PHYS_PROCESSOR(R9), R7
01E4 001A*
01E6 4D95      LD         VPT.VP.NEXT_READY_VP(R9), #NIL
01E8 001C*
01EA FFFF
01EC 4D95      LD         VPT.VP.MSG_LIST(R9), #NIL
01EE 001E*
01F0 FFFF

               ! CHECK EXTERNAL VP FLAG !
01F2 0B06      CP         R6, #ON
01F4 FFFF
                IF EQ !EXTERNAL VP!
01F6 5E0E       THEN  ! VP IS TC VISIBLE !
01F8 0210'
01FA 6108        LD       R8, NEXT_EXT_VP
01FC 0002'
                 ! INSERT ENTRY IN EXTERNAL VP LIST !
01FE 6F89        LD       EXT_VP_LIST(R8), R9
```

```
0200 0000*
0202 6F98        LD       VPT.VP.EXT_ID(R9), R8
0204 0020*
0206 A981        INC      R8, #2
0208 6F08        LD       NEXT_EXT_VP, R8
020A 0002'
020C 5E08       ELSE  !VP BOUND TO KERNEL PROCESS!
020E 0216'
0210 4D05        LD       VPT.VP.EXT_ID, #NIL
0212 0020*
0214 FFFF
               FI
0216 A19A      LD         R10, R9
0218 010A      ADD        R10, #SIZEOF VP_TABLE
021A 0020
021C 6F0A      LD         NEXT_AVAIL_VP, R10
021E 0000'
0220 9E08      RET
0222         END UPDATE_VP_TABLE
          END BOOTSTRAP_LOADER
```

APPENDIX F - LIBRARY FUNCTION LISTINGS

```
Z8000ASM  2.02
LOC     OBJ CODE     STMT SOURCE STATEMENT


                 LIBRARY_FUNCTION  MODULE

         $LISTON $TTY

         CONSTANT
           KERNEL_FCW       := %5000
           STACK_SEG_SIZE   := %100
           STACK_BASE       := STACK_SEG_SIZE-%10
           STATUS_REG_BLOCK:= STACK_SEG_SIZE-%10
           INTERRUPT_FRAME  := STACK_BASE-4
           INTERRUPT_REG    := INTERRUPT_FRAME-34
           N_S_P            := INTERRUPT_REG-2
           NIL              := %FFFF
```

```
        $SECTION LIB_PROC
        GLOBAL

0000            LIST_INSERT              PROCEDURE
               !*********************************
                * INSERTS OBJECTS INTO A LIST   *
                * BY ORDER OF PRIORITY AND SETS *
                * ITS STATE                     *
                *********************************
                * REGISTER USE:                 *
                *  PARAMETERS:                  *
                *   R2: OBJECT ID               *
                *   R3: HEAD_OF_LIST_PTR ADDR   *
                *   R4: NEXT_OBJ_PTR ADDR       *
                *   R5: PRIORITY_PTR ADDR       *
                *   R6: STATE_PTR ADDR          *
                *   R7: OBJECT STATE            *
                *  LOCAL VARIABLES:             *
                *   R8: HEAD_OF_LIST_PTR        *
                *   R9: NEXT_OBJ_PTR            *
                *   R10: CURRENT_OBJ PRIORITY   *
                *   R11: NEXT_OBJ PRIORITY      *
                *********************************!

                ENTRY
                ! GET FIRST OBJECT IN LIST !
 0000 2138      LD              R8, @R3
 0002 0B08      CP              R8, #NIL
 0004 FFFF
 0006 5E0E      IF EQ !LIST IS EMPTY! THEN
 0008 0018'
                 ! PLACE OBJ AT HEAD OF LIST !
 000A 2F32       LD             @R3, R2
 000C 7449       LDA            R9, R4(R2)
 000E 0200
 0010 0D95       LD             @R9, #NIL
 0012 FFFF
 0014 5E08      ELSE
 0016 005A'
                 ! COMPARE OBJ PRI WITH LIST HEAD PRI !
 0018 715A       LD             R10, R5(R2) !OBJ PRI!
 001A 0200
 001C 715B       LD             R11, R5(R8) !HEAD PRI!
 001E 0800
 0020 8BBA       CP             R10, R11
 0022 5E02       IF GT !OBJ PRI>HEAD PRI! THEN
 0024 0030'
 0026 2F32        LD            @R3, R2  !PUT AT FRONT!
 0028 7348        LD            R4(R2), R8
 002A 0200
 002C 5E08       ELSE ! INSERT IN BODY OF LIST !
```

AD-A102 308 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
IMPLEMENTATION OF PROCESS MANAGEMENT FOR A SECURE ARCHIVAL STOR--ETC(U)
MAR 81 A R STRICKLER

UNCLASSIFIED NL

3 of 3
AD
A102308

END
DATE
FILMED
8 81
DTIC

```
002E 005A'

               SEARCH_LIST:
                 DO
0030 0B08          CP          R8, #NIL
0032 FFFF
0034 5E0E          IF EQ !END OF LIST! THEN
0036 003C'
0038 5E08           EXIT FROM SEARCH_LIST
003A 0052'
                   FI
003C 715B          LD          R11, R5(R8) !GET NEXT PRI!
003E 0800
0040 8BBA          CP          R10, R11
0042 5E02          IF GT !CURRENT PRI>NEXT PRI! THEN
0044 004A'
0046 5E08           EXIT FROM SEARCH_LIST
0048 0052'
                   FI

                   ! GET NEXT OBJ !
004A A189          LD          R9, R8
004C 7148          LD          R8, R4(R9)
004E 0900
0050 E8EF        OD   ! END SEARCH_LIST !

                 ! INSERT IN LIST !
0052 7348        LD            R4(R2), R8
0054 0200
0056 7342        LD            R4(R9), R2
0058 0900
                FI
               FI

               ! SET OBJECT'S STATE !
005A 7367      LD              R6(R2), R7
005C 0200
005E 9E08      RET
0060          END LIST_INSERT
```

```
ØØ6Ø              CREATE_STACK          PROCEDURE
                  !********************************
                  * INITIALIZES KERNEL STACK      *
                  * SEGMENT FOR PROCESSES         *
                  *********************************
                  * REGISTER USE:                 *
                  *  PARAMETERS:                  *
                  *   RØ: ARGUMENT POINTER        *
                  *   (INCLUDES:FCW,IC,NSP, AND   *
                  *    RETURN POINT. SEE LOCAL    *
                  *    VARIABLES BELOW.)          *
                  *   R1: TOP OF STACK            *
                  *   R2-R14: INITIAL REGISTER    *
                  *    STATES. (NOTE: IN DEMO, NO*
                  *    SPECIFIC INITIAL REGISTER *
                  *    VALUES ARE SET, EXCEPT R13*
                  *    (USER ID) FOR USER PRO-   *
                  *    CESSES.)                   *
                  *********************************
                  *  LOCAL VARIABLES              *
                  *  (FROM ARGUMENTS STORED ON    *
                  *   STACK.)                     *
                  *   R3: FCW                     *
                  *   R4: PROCESS ENTRY POINT(IC)*
                  *   R5: NSP                     *
                  *   R6: PREEMPT RETURN POINT    *
                  ********************************!

                  ENTRY
ØØ6Ø 93FØ         PUSH      @R15, RØ !SAVE ARGUMENT PTR!
ØØ62 ADFØ         EX        RØ, R15  !SAVE SP!
ØØ64 341F         LDA       R15, R1(#INTERRUPT_REG)
ØØ66 ØØCA
ØØ68 1CF9         LDM       @R15, R1, #16  !INITIAL REG. VALUES!
ØØ6A Ø1ØF
                  ! NOTE: ONLY REGISTERS R2-R14 MAY CONTAIN
                    INITIALIZATION VALUES !
ØØ6C A1ØF         LD        R15, RØ  !RESTORE SP!
ØØ6E 97FØ         POP       RØ, @R15 !RESTORE ARGUMENT PTR!
ØØ7Ø A1FE         LD        R14, R15 !SAVE CALLER RETURN POINT!
ØØ72 A1ØF         LD        R15, RØ  !GET ARGUMENT PTR!
ØØ74 1CF1         LDM       R3, @R15, #4 !LOAD ARGUMENTS!
ØØ76 Ø3Ø3
ØØ78 341F         LDA       R15, R1(#INTERRUPT_FRAME)
ØØ7A ØØEC
ØØ7C 1CF9         LDM       @R15, R3, #2 !INIT IRET FRAME!
ØØ7E Ø3Ø1
ØØ8Ø 341F         LDA       R15, R1(#N_S_P)
ØØ82 ØØC8
ØØ84 2FF5         LD        @R15, R5  !SET NSP!
ØØ86 Ø3ØF         SUB       R15, #2
```

```
0088 0002
008A 2FF6       LD          @R15, R6 !PREEMPT RET POINT!
008C 3418       LDA         R8, R1(#STACK_BASE)
008E 00F0
                ! INITIALIZE STATUS REGISTER BLOCK !
0090 2100       LD          R0, #KERNEL_FCW
0092 5000
0094 1C89       LDM         @R8, R15, #2  !SAVE SP & FCW!
0096 0F01
0098 A1EF       LD          R15, R14  !RESTORE RETURN POINT!
009A 9E08       RET
009C          END CREATE_STACK
          END LIBRARY_FUNCTION
```

APPENDIX G - INNER TRAFFIC CONTOLLER LISTINGS

```
Z8000ASM  2.02
LOC    OBJ CODE    STMT SOURCE STATEMENT

               INNER_TRAFFIC_CONTROL MODULE

          $LISTON $TTY

      !**1. GETWORK:
              A. NORMAL ENTRY DOES NOT SAVE REGISTERS.
              ( THIS IS A FUNCTION OF THE GATEKEEPER ).
              B. R14 IS AN INPUT PARAMETER TO GETWORK THAT
               SIMULATES INFO THAT WILL EVENTUALLY BE ON
               THE MMU HARDWARE.  THIS REGISTER MUST BE
               ESTABLISHED AS A DBR BY ANY PROCEDURE
               INVOKING GETWORK.
            C. THE PREEMPT INTERRUPT ENTRY HANDLER DOES
               NOT USE THE GATEKEEPER AND MUST PERFORM
               FUNCTIONS NORMALLY ACCOMPLISHED BY IT
               PRIOR TO NORMAL ENTRY AND EXIT.
             ( SAVE/RESTORE: REGS, NSP; UNLOCK VPT, TEST INT)

         2. GENERAL:
            A. ALL VIOLATIONS OF VIRTUAL MACHINE INSTRUCTIONS
               ARE CONSIDERED ERROR CONDITIONS AND WILL RETURN
               SYSTEM TO THE MONITOR WITH AN ERROR CODE IN R0
               AND THE PC VALUE IN R1.
            B. ITC PROCEDURES CALLING GETWORK PASS DBR
               (REGISTER R14) AND LOGICAL CPU NUMBER
               (REGISTER R13) AS INPUT PARAMETERS.
               (INCLUDES: SIGNAL, WAIT, SWAP_VDBR,
                PHYS_PREEMPT_HANDLER, AND IDLE).  !

            CONSTANT
                 ! *********** ERROR CODES *********** !
              U_L        := 0     ! UNAUTHORIZED LOCK !
              M_L_EM     := 1     ! MESSAGE LIST EMPTY !
              M_L_ER     := 2     ! MESSAGE LIST ERROR !
              R_L_E      := 3     ! READY LIST EMPTY !
              M_L_O      := 4     ! MESSAGE LIST OVERFLOW !
              S_N_A      := 5     ! SWAP NOT ALLOWED !
              V_I_E      := 6     ! VP INDEX ERROR !
              M_U        := 7     ! MMU UNAVAILABLE !

                 ! ********* SYSTEM PARAMETERS ********* !
                 NR_SDR          := 64   !LONG WORDS!
                 NR_CPU          := 2
                 NR_VP           := NR_CPU*4
                 NR_AVAIL_VP     := NR_CPU*2
```

```
     MAX_DBR_NR        := 10  !PER CPU!
     STACK_SEG         := 1
     PRDS_SEG          := 0
     STACK_SEG_SIZE    := %100

     ! ***** OFFSETS IN STACK SEG ***** !
     STACK_BASE        := STACK_SEG_SIZE-%10
     STATUS_REG_BLOCK:= STACK_SEG_SIZE-%10
     INTERRUPT_FRAME   := STACK_BASE-4
     INTERRUPT_REG     := INTERRUPT_FRAME-34
     N_S_P             := INTERRUPT_REG-2
     F_C_W             := STACK_SEG_SIZE-%E

     ON       := %FFFF
     OFF      := 0
     RUNNING  := 0
     READY    := 1
     WAITING  := 2
     NIL      := %FFFF
     INVALID  := %EEEE
     MONITOR   := %A900        ! HBUG ENTRY !
     KERNEL_FCW   := %5000
     AVAILABLE    := 0
     ALLOCATED    := %FF

  TYPE
     MESSAGE   ARRAY [16   BYTE]
     ADDRESS   WORD
     VP_INDEX          INTEGER
     MSG_INDEX         INTEGER

     SEG_DESC_REG   RECORD
                   [
                    BASE       ADDRESS
                    ATTRIBUTES         BYTE
                    LIMITS             BYTE
                   ]

     MMU               ARRAY[NR_SDR SEG_DESC_REG]

     MSG_TABLE RECORD
      [ MSG            MESSAGE
        SENDER         VP_INDEX
        NEXT_MSG       MSG_INDEX
        FILLER         ARRAY [6, WORD]
      ]
```

```
                  VP_TABLE RECORD
                      [ DBR    ADDRESS
                        PRI            WORD
                        STATE          WORD
                        IDLE_FLAG      WORD
                        PREEMPT        WORD
                        PHYS_PROCESSOR  WORD
                        NEXT_READY_VP  VP_INDEX
                        MSG_LIST       MSG_INDEX
                        EXT_ID         WORD
                        FILLER_1       ARRAY [7, WORD]
                      ]

                  EXTERNAL
                    LIST_INSERT        PROCEDURE

                  GLOBAL
                    BOOTSTRAP_ENTRY    LABEL

                  $SECTION ITC_DATA

0000                VPT       RECORD
                      [ LOCK          WORD
                        RUNNING_LIST  ARRAY[NR_CPU WORD]
                        READY_LIST    ARRAY[NR_CPU WORD]
                        FREE_LIST     MSG_INDEX
                        VIRT_INT_VEC  ARRAY[1, ADDRESS]
                        FILLER_2      WORD
                        VP         ARRAY [NR_VP, VP_TABLE]
                        MSG_Q         ARRAY [NR_VP, MSG_TABLE]
                      ]
0210                EXT_VP_LIST  ARRAY[NR_AVAIL_VP   WORD]

                 $SECTION MMU_DATA

0000                MMU_IMAGE        RECORD
                        [
                          MMU_STRUCTURE        ARRAY[MAX_DBR_NR   MMU]
                        ]
0A00                NEXT_AVAIL_MMU  ARRAY[MAX_DBR_NR  BYTE]
0A0A                PRDS     RECORD
                         [PHYS_CPU_ID WORD
                          LOG_CPU_ID  INTEGER
                          VP_NR       WORD
                          IDLE_VP     VP_INDEX]
                 $SECTION ITC_INT_PROC
                   INTERNAL
0000               GETWORK                   PROCEDURE
                   !*******************************
                    * SWAPS VIRTUAL PROCESSORS    *
                    * ON PHYSICAL PROCESSOR.      *
                    *******************************
                    * PARAMETERS:                 *
                    *  R13: LOGICAL CPU #         *
                    * REGISTER USE:               *
                    *  STATUS REGISTERS           *
                    *   R14: DBR (SIMULATION)     *
                    *   R15: STACK_POINTER        *
                    *  LOCAL VARIABLES:           *
                    *   R1: READY_VP (NEW)        *
                    *   R2: CURRENT_VP (OLD)      *
                    *   R3: FLAG CONTROL WORD     *
                    *   R4: STACK_SEG BASE ADDR   *
                    *   R5: STATUS_REG_BLOCK ADDR *
                    *   R6: NORMAL STACK POINTER  *
                    *******************************!
                    ENTRY

                    ! GET STACK BASE !
0000 31E4           LD          R4, R14(#STACK_SEG*4)
0002 0004
0004 3445           LDA         R5, R4(#STATUS_REG_BLOCK)
0006 00F0
                    ! * * SAVE SP * * !
0008 2F5F           LD          @R5, R15
                    ! * * SAVE FCW * * !
000A 7D32           LDCTL       R3, FCW
000C 3343           LD          R4(#F_C_W), R3
000E 00F2

                BOOTSTRAP_ENTRY:          ! GLOBAL LABEL !
                    ! GET READY_VP LIST !
0010 61D1           LD          R1, VPT.READY_LIST(R13)
0012 0006'

                   SELECT_VP:
                    DO  ! UNTIL ELGIBLE READY_VP FOUND !
0014 4D11             CP VPT.VP.IDLE_FLAG(R1), #ON
0016 0016'
0018 FFFF
001A 5E0E             IF EQ  ! VP IS IDLE ! THEN
001C 0030'
001E 4D11               CP VPT.VP.PREEMPT(R1), #ON
0020 0018'
0022 FFFF
0024 5E0E               IF EQ  ! PREEMPT INTERRUPT IS ON !  THEN
```

```
0026 002C'
0028 5E08           EXIT FROM SELECT_VP
002A 003C'
                  FI
002C 5E08         ELSE ! VP NOT IDLE !
002E 0034'
0030 5E08           EXIT FROM SELECT_VP
0032 003C'
                  FI
                  ! GET NEXT READY_VP !
0034 6113         LD  R3, VPT.VP.NEXT_READY_VP(R1)
0036 001C'
0038 A131         LD  R1, R3
003A E8EC        OD

                 ! NOTE: THE READY_LIST WILL NEVER BE EMPTY SINCE
                    THE IDLE VP, WHICH IS THE LOWEST PRI VP,
                    WILL NEVER BE REMOVED FROM THE LIST.
                    IT WILL RUN ONLY IF ALL OTHER READY VP'S ARE
                    IDLING OR IF THERE ARE NO OTHER VP'S ON
                    THE READY_LIST. ONCE SCHEDULED, IT
                    WILL RUN UNTIL RECEIVING A HDWE INTERRUPT. !

                 ! NOTE: R14 IS USED AS DBR HERE. WHEN MMU
                    IS AVAILABLE THIS SERIES OF SAVE AND LOAD
                    INSTRUCTIONS WILL BE REPLACED BY SPECIAL I/O
                    INSTRUCTIONS TO THE MMU. !
                 ! PLACE NEW_VP IN RUNNING STATE !
003C 4D15        LD   VPT.VP.STATE(R1), #RUNNING
003E 0014'
0040 0000
0042 6FD1        LD   VPT.RUNNING_LIST(R13), R1
0044 0002'
                 ! * * SWAP DBR * * !
0046 611E        LD   R14, VPT.VP.DBR(R1)
0048 0010'

                 ! LOAD NEW_VP SP !
004A 31E4        LD   R4, R14(#STACK_SEG*4)
004C 0004
004E 3445        LDA  R5, R4(#STATUS_REG_BLOCK)
0050 00F0
0052 215F        LD   R15, @R5

                 ! * * LOAD NEW FCW * * !
0054 3143        LD   R3, R4(#F_C_W)
0056 00F2
0058 7D3A        LDCTL  FCW, R3
005A 9E08        RET
005C            END GETWORK
```

```
005C            ENTER_MSG_LIST           PROCEDURE
               !*********************************
                * INSERTS POINTER TO MESSAGE    *
                * FROM CURRENT_VP TO SIGNALED_VP*
                * IN FIFO MSG_LIST              *
                *********************************
                * REGISTER USE:                 *
                *  PARAMETERS:                  *
                *   R8(R9):MSG (INPUT)          *
                *   R1: SIGNALED_VP (INPUT)     *
                *   R13: LOGICAL CPU NUMBER     *
                *  LOCAL VARIABLES:             *
                *   R2: CURRENT_VP              *
                *   R3: FIRST_FREE_MSG          *
                *   R4: NEXT_FREE_MSG           *
                *   R5: NEXT_Q_MSG              *
                *   R6: PRESENT_Q_MSG           *
                *********************************!
                ENTRY
005C 61D2        LD  R2, VPT.RUNNING_LIST(R13)
005E 0002'

                 ! GET FIRST MSG FROM FREE_LIST !
0060 6103        LD  R3, VPT.FREE_LIST
0062 000A'

                     ! * * * * DEBUG * * * * !
0064 0B03            CP R3, #NIL
0066 FFFF
0068 5E0E            IF EQ THEN
006A 0078'
006C 7601             LDA R1, $
006E 006C'
0070 2100             LD R0, #M_L_O! MESSAGE LIST OVERFLOW !
0072 0004
0074 5F00             CALL MONITOR
0076 A900
                     FI
                     ! * * * END DEBUG * * * !

0078 6134        LD  R4, VPT.MSG_Q.NEXT_MSG(R3)
007A 0122'
007C 6F04        LD  VPT.FREE_LIST, R4
007E 000A'
                 ! INSERT MESSAGE LIST INFORMATION !
0080 763A        LDA       R10,VPT.MSG_Q.MSG(R3)
0082 0110'
0084 2107        LD        R7,#SIZEOF MESSAGE
0086 0010
0088 BA81        LDIRB     @R10,@R8,R7
008A 07A0
```

```
008C 6F32,    LD  VPT.MSG_Q.SENDER(R3), R2
008E 0120'

              ! INSERT MSG IN MSG_LIST !
0090 6115     LD  R5, VPT.VP.MSG_LIST(R1)
0092 001E'

0094 0B05     CP  R5, #NIL
0096 FFFF
0098 5E0E     IF EQ  ! MSG LIST IS EMPTY !  THEN
009A 00A4'
               ! INSERT MSG AT TOP OF LIST !
009C 6F13      LD VPT.VP.MSG_LIST(R1), R3
009E 001E'

00A0 5E08     ELSE  ! INSERT MSG IN LIST !
00A2 00BC'
              MSG_Q_SEARCH:
              DO  ! WHILE NOT END OF LIST !
00A4 0B05      CP        R5, #NIL
00A6 FFFF
00A8 5E0E      IF EQ  ! END OF LIST !  THEN
00AA 00B0'
00AC 5E08       EXIT FROM MSG_Q_SEARCH
00AE 00B8'
               FI

               ! GET NEXT LINK !
00B0 A156      LD        R6, R5
00B2 6165      LD        R5, VPT.MSG_Q.NEXT_MSG(R6)
00B4 0122'
00B6 E8F6     OD
              ! INSERT MSG IN LIST !
00B8 6F63     LD         VPT.MSG_Q.NEXT_MSG(R6), R3
00BA 0122'
              FI
00BC 6F35     LD          VPT.MSG_Q.NEXT_MSG(R3), R5
00BE 0122'
00C0 9E08     RET
00C2       END ENTER_MSG_LIST
```

```
00C2            GET_FIRST_MSG                 PROCEDURE
               !*************************************
                * REMOVES MSG FROM MSG_LIST          *
                * AND PLACES ON FREE LIST.           *
                * RETURNS SENDER'S MSG AND           *
                * VP_ID                              *
                **************************************
                *REGISTER USE:                       *
                * PARAMETERS:                        *
                *  R8(R9): MSG POINTER (INPUT)       *
                *  R13: LOGICAL CPU NUMBER (INPUT)   *
                *  R1: SENDER VP (RETURNED)          *
                * LOCAL VARIABLES                    *
                *  R2: CURRENT_VP                    *
                *  R3: FIRST_MSG                     *
                *  R4: NEXT_MSG                      *
                *  R5: NEXT_FREE_MSG                 *
                *  R6: PRESENT_FREE_MSG              *
                **************************************!
                ENTRY
00C2 61D2       LD         R2, VPT.RUNNING_LIST(R13)
00C4 0002'

                ! REMOVE FIRST MSG FROM MSG_LIST !
00C6 6123       LD         R3, VPT.VP.MSG_LIST(R2)
00C8 001E'

                           ! * * * * DEBUG * * * * !
00CA 0B03                  CP R3, #NIL
00CC FFFF
00CE 5E0E                  IF EQ THEN
00D0 00DE'
00D2 2100                   LD R0, #M_L_EM  ! MSG LIST EMPTY !
00D4 0001
00D6 7601                   LDA R1, $
00D8 00D6'
00DA 5F00                   CALL MONITOR
00DC A900
                           FI
                           ! * * * END DEBUG * * * !
00DE 6134       LD         R4, VPT.MSG_Q.NEXT_MSG(R3)
00E0 0122'
00E2 6F24       LD         VPT.VP.MSG_LIST(R2), R4
00E4 001E'
                          ! INSERT MESSAGE IN FREE_LIST !
00E6 6105       LD         R5, VPT.FREE_LIST
00E8 000A'
00EA 0B05       CP         R5, #NIL
00EC FFFF
00EE 5E0E       IF EQ    ! FREE_LIST IS EMPTY !  THEN
00F0 0100'
```

```
                    ! INSERT AT TOP OF LIST !
00F2 6F03           LD          VPT.FREE_LIST, R3
00F4 000A'
00F6 4D35           LD          VPT.MSG_Q.NEXT_MSG(R3), #NIL
00F8 0122'
00FA FFFF
00FC 5E08         ELSE   ! INSERT IN LIST !
00FE 011C'
                FREE_Q_SEARCH:
                    DO

0100 0B05            CP         R5, #NIL
0102 FFFF
0104 5E0E            IF EQ   ! END OF LIST !   THEN
0106 010C'
0108 5E08              EXIT FROM FREE_Q_SEARCH
010A 0114'
                     FI
                     ! GET NEXT MSG !
010C A156            LD         R6, R5
010E 6165            LD         R5, VPT.MSG_Q.NEXT_MSG(R6)
0110 0122'
0112 E8F6           OD

                  ! INSERT IN LIST !
0114 6F63           LD          VPT.MSG_Q.NEXT_MSG(R6), R3
0116 0122'
0118 6F35           LD          VPT.MSG_Q.NEXT_MSG(R3), R5
011A 0122'
                  FI
                  ! GET MESSAGE INFORMATION:
                    (RETURNS R1: SENDING_VP)  !
011C 6131         LD            R1, VPT.MSG_Q.SENDER(R3)
011E 0120'
0120 763A         LDA           R10,VPT.MSG_Q.MSG(R3)
0122 0110'
0124 2107         LD            R7,#SIZEOF MESSAGE
0126 0010
0128 BAA1         LDIRB         @R8,@R10,R7
012A 0780
012C 9E08         RET
012E            END GET_FIRST_MSG
```

```
                    !  * * INNER TRAFFIC CONTROL ENTRY POINTS * * !

                    ! NOTE: ALL INTERRUPTS MUST BE MASKED WHENEVER
                      THE VPT IS LOCKED.  THIS IS TO PREVENT AN
                      EMBRACE FROM OCCURRING SHOULD AN INTERRUPT
                      OCCUR WHILE THE VPT IS LOCKED. !

                    GLOBAL
                    $SECTION ITC_GLB_PROC

                    PREEMPT_RET LABEL
                    KERNEL_EXIT LABEL
0000                CREATE_INT_VEC         PROCEDURE
                   !*******************************
                    * CREATES ENTRY IN VIRTUAL INT-*
                    * ERRUPT VECTOR WITH ADDRESS   *
                    * OF THE VIRTUAL INTERRUPT HAN-*
                    * DLER.                        *
                    ********************************
                    * PARAMETERS:                  *
                    *  R1: VIRTUAL INTERRUPT #     *
                    *  R2: INTERRUPT HANDLER ADDR  *
                    ********************************!

                    ENTRY
                     ! COMPUTE OFFSET IN VIRTUAL
                       INTERRUPT VECTOR !
0000 1900            MULT      RR0, #SIZEOF ADDRESS
0002 0002
                     ! SAVE ADDRESS OF VIRTUAL INTERRUPT
                       HANDLER IN INTERRUPT VECTOR !
0004 6F12            LD        VPT.VIRT_INT_VEC(R1), R2
0006 000C
0008 9E08            RET
000A               END CREATE_INT_VEC
```

```
000A          GET_DBR_ADDR           PROCEDURE
             !*********************************
              * CALCULATES DBR ADDRESS FROM   *
              * DBR NUMBER                    *
              *********************************
              * REGISTER USE:                 *
              *  PARAMETERS:                  *
              *   R0: DBR #                   *
              *  RETURNS:                     *
              *   R1: DBR ADDRESS             *
              *********************************!
              ENTRY
               ! GET BASE ADDRESS OF MMU IMAGE !
000A 7601      LDA        R1, MMU_IMAGE
000C 0000'
               ! ADD DBR HANDLE (OFFSET) TO MMU BASE
                 ADDRESS TO OBTAIN DBR ADDRESS !
000E 8101      ADD        R1, R0
0010 9E08      RET
0012          END GET_DBR_ADDR
```

```
0012          ALLOCATE_MMU              PROCEDURE
             !*********************************
              * ALLOCATES NEXT AVAILABLE MMU *
              * IMAGE AND CREATES PRDS ENTRY *
              ********************************
              * REGISTER USE:                *
              *  RETURNS:                    *
              *   R0: DBR #                  *
              *  LOCAL VARIABLES:            *
              *   R1: SEGMENT #              *
              *   R2: PRDS ADDRESS           *
              *   R3: PRDS ATTRIBUTES        *
              *   R4: PRDS LIMITS            *
              ********************************!
              ENTRY
               ! GET NEXT AVAILABLE DBR # !
0012 8D08      CLR        R0
0014 8D18      CLR        R1
               ! NOTE: THE FOLLOWING IS A SAFE SECUENCE
                 AS NEXT_AVAIL_MMU AND MMU ARE CPU LOCAL!
          GET_DBR:
               DO
0016 4C11        CPB     NEXT_AVAIL_MMU(R1), #AVAILABLE
0018 0A00'
001A 0000
                 IF EQ  !MMU ENTRY IS AVAILABLE!
001C 5E0E          THEN
001E 002E'
0020 4C15           LDB  NEXT_AVAIL_MMU(R1), #ALLOCATED
0022 0A00'
0024 FFFF
0026 5E08           EXIT FROM GET_DBR
0028 004A'
002A 5E08          ELSE  !CURRENT ENTRY IS ALLOCATED!
002C 0048'
002E A910           INC  R1, #1
0030 0100           ADD  R0, #SIZEOF MMU
0032 0100
                     ! * * * * DEBUG * * * * !
0034 0B01            CP  R1, #MAX_DBR_NR
0036 000A
0038 5E0E            IF EQ THEN
003A 0048'
003C 2100              LD        R0, #M_U  !MMU UNAVAILABLE!
003E 0007
0040 7601              LDA       R1, $
0042 0040'
0044 5F00              CALL      MONITOR
0046 A900
                     FI
                     ! * * * END DEBUG * * * !
```

```
                  FI
0048 E8E6       OD

004A 2101       LD          R1, #PRDS_SEG    ! SEGMENT NO. !
004C 0000
004E 7602       LDA         R2, PRDS         ! PRDS ADDR    !
0050 0A0A'
0052 2103       LD          R3, #1 ! READ ATTR    !
0054 0001
0056 2104       LD          R4, #((SIZEOF PRDS)-1)/256
0058 0000
                ! PRDS LIMITS !


               ! CREATE PRDS ENTRY IN MMU IMAGE !
005A 5F00       CALL        UPDATE_MMU_IMAGE  !(R1: SEGMENT #
005C 0060'
                                                R2: SEG ADDRESS
                                                R3: ATTRIBUTES
                                                R4: SEG LIMITS)!
005E 9E08       RET
0060          END ALLOCATE_MMU
```

```
0060         UPDATE_MMU_IMAGE        PROCEDURE
            !********************************
             * CREATES SEGMENT DESCRIPTOR    *
             * ENTRY IN MMU IMAGE            *
             *********************************
             * REGISTER USE:                 *
             *  PARAMETERS:                  *
             *   R0: DBR #                   *
             *   R1: SEGMENT #               *
             *   R2: SEGMENT ADDRESS         *
             *   R3: SEGMENT ATTRIBUTES      *
             *   R4: SEGMENT LIMITS          *
             *  LOCAL VARIABLES:             *
             *   R10: MMU BASE ADDRESS       *
             *   R13: OFFSET VARIABLE        *
             ********************************!
             ENTRY
0060 210A     LD  R10, #MMU_IMAGE ! MMU BASE ADDRESS !
0062 0000'
0064 810A     ADD  R10, R0
0066 210D     LD  R13, #SIZEOF SEG_DESC_REG
0068 0004
006A 991C     MULT  RR12, R1 ! COMPUTE SEG_DESC OFFSET !
006C 81DA     ADD  R10, R13 !ADD OFFSET TO BASE ADDRESS!
              ! INSERT DESCRIPTOR DATA !
006E 2FA2     LD  @R10, R2
0070 A9A1     INC  R10, #2
0072 0DA8     CLR  @R10
0074 2EAC     LDB  @R10, RL4
0076 A9A0     INC  R10, #1
0078 20AC     LDB  RL4, @R10
007A 0A0B     CPB  RL3, #%(2)00001000 ! EXECUTE !
007C 0808
007E 5E0E     IF  EQ  THEN
0080 008A'
0082 060C         ANDB  RL4, #%(2)11110111  ! EXECUTE MASK !
0084 F7F7
0086 5E08     ELSE
0088 008E'
008A 060C         ANDB  RL4, #%(2)11111110  ! READ MASK !
008C FEFE
              FI
008E 84BC     ORB  RL4, RL3
0090 2EAC     LDB  @R10, RL4
0092 9E08     RET
0094        END UPDATE_MMU_IMAGE
```

```
0094           WAIT                           PROCEDURE
              !*************************************
               * INTRA_KERNEL SYNC/COM PRIMATIVE *
               * INVOKED BY KERNEL PROCESSES     *
               ***********************************
               * PARAMETERS                      *
               *  R8(R9): MSG POINTER (INPUT)    *
               *  R1: SENDING_VP (RETURN)        *
               * GLOBAL VARIABLES                *
               *  R14: DBR (PARAM TO GETWORK)    *
               * LOCAL VARIABLES                 *
               *  R2: CURRENT_VP (RUNNING)       *
               *  R3: NEXT_READY_VP              *
               *  R4: LOCK_ADDRESS               *
               *  R13: LOGICAL CPU NUMBER        *
               ***********************************!
              ENTRY
                ! MASK INTERRUPTS !
0094 7C01       DI    VI
                ! LOCK VPT !
0096 7604       LDA         R4, VPT.LOCK
0098 0000'
009A 5F00       CALL        SPIN_LOCK  ! (R4:^VPT.LOCK) !
009C 0282'
              ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
              ! GET CPU NUMBER !
009E 5F00       CALL        GET_CPU_NO  !RETURNS:
00A0 02C8'
                                           R1:CPU #
                                           R2:# VP'S!
00A2 A11D       LD          R13, R1

00A4 61D2       LD          R2, VPT.RUNNING_LIST(R13)
00A6 0002'
00A8 6123       LD          R3, VPT.VP.NEXT_READY_VP(R2)
00AA 001C'

00AC 4D21       CP          VPT.VP.MSG_LIST(R2), #NIL
00AE 001E'
00B0 FFFF
00B2 5E0E        IF EC ! CURRENT VP'S MSG LIST IS EMPTY ! THEN
00B4 00EA'
                 ! REMOVE CURRENT_VP FROM READY_LIST !
                            ! * * * * DEBUG * * * * !
00B6 0B03                   CP      R3, #NIL
00B8 FFFF
00BA 5E0E                   IF EQ  THEN
00BC 00CA'
00BE 2100                    LD  R0, #R_L_E  ! READY LIST EMPTY !
00C0 0003
00C2 7601                    LDA R1, $
```

```
00C4 00C2'
00C6 5F00                 CALL MONITOR
00C8 A900
                        FI
                        ! * * * END DEBUG * * * !

00CA 6FD3      LD       VPT.READY_LIST(R13), R3
00CC 0006'
00CE 4D25      LD       VPT.VP.NEXT_READY_VP(R2), #NIL
00D0 001C'
00D2 FFFF

               ! PUT IT IN WAITING STATE !
00D4 4D25      LD VPT.VP.STATE(R2), #WAITING
00D6 0014'
00D8 0002
               ! SET DBR !
00DA 612E      LD        R14, VPT.VP.DBR(R2)
00DC 0010'
               ! SCHEDULE FIRST ELGIBLE READY VP !
00DE 93F8      PUSH       @R15,R8
               ! SAVE LOGICAL CPU # !
00E0 93FD      PUSH    @R15, R13
00E2 5F00                 CALL     GETWORK  !R13:CPU #
00E4 0000'
                                           R14:DBR!
               ! RESTORE CPU # !
00E6 97FD      POP      R13, @R15
00E8 97F8      POP         R8,@R15
             FI
             ! GET FIRST MSG ON CURRENT VP'S MSG LIST !
00EA 5F00    CALL GET_FIRST_MSG  ! COPIES MSG IN MSG ARRAY!
00EC 00C2'
                                 ! R13: LOGICAL CPU # !
                                 !RETURNS R1:SENDER_VP !

             ! UNLOCK VPT !
00EE 4D08    CLR  VPT.LOCK
00F0 0000'
             ! UNMASK VECTORED INTERRUPTS !
00F2 7C05    EI    VI

             ! RETURN: R1:SENDER_VP !
00F4 9E08    RET
00F6      END WAIT
```

```
00F6         SIGNAL                           PROCEDURE
            !***********************************
             * INTRA KERNEL SYNC /COM PRIMATIVE *
             * INVOKED BY KERNEL PROCESSES      *
             ************************************
             * REGISTER USE:                    *
             *  PARAMETERS:                     *
             *   R8(R9): MSG POINTER (INPUT)    *
             *   R1: SIGNALED VP_ID (INPUT)     *
             * GLOBAL VARIABLES                 *
             *  R13: CPU # (PARAM TO GETWORK)   *
             *  R14: DBR (PARAM TO GETWORK)     *
             *  LOCAL VARIABLES:                *
             *   R1: SIGNALED VP                *
             *   R2: CURRENT_VP                 *
             *   R4: VPT.LOCK ADDRESS           *
             ************************************!
             ENTRY
              ! SAVE VP ID !
00F6 93F1     PUSH    @R15, R1
              ! MASK INTERRUPTS !
00F8 7C01     DI    VI
              ! LOCK VPT !
00FA 7604     LDA         R4, VPT.LOCK
00FC 0000'
00FE 5F00     CALL        SPIN_LOCK  ! (R4:^VPT.LOCK) !
0100 0282'
             !NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP. !
              ! GET LOGICAL CPU # !
0102 5F00     CALL        GET_CPU_NO !RETURNS:
0104 02C8'
                                      R1:CPU #
                                      R2:# VP'S!
0106 A11D     LD          R13, R1
              ! RESTORE VP ID !
0108 97F1     POP         R1, @R15

              ! PLACE MSG IN SIGNALED_VP'S MSG_LIST !
010A 5F00     CALL ENTER_MSG_LIST !(R8:MSG POINTER
010C 005C'                            R1:SIGNALED_VP
                                      R13:LOGICAL CPU #)!

010E 4D11     CP          VPT.VP.STATE(R1), #WAITING
0110 0014'
0112 0002
0114 5E0E     IF EQ  ! SIGNALED_VP IS WAITING !  THEN
0116 0148'

                ! WAKE IT UP AND MAKE IT READY !
0118 A112       LD        R2, R1
011A 76D3       LDA       R3, VPT.READY_LIST(R13)
```

```
011C 0006'
011E 7604       LDA     R4, VPT.VP.NEXT_READY_VP
0120 001C'
0122 7605       LDA     R5, VPT.VP.PRI
0124 0012'
0126 7606       LDA     R6, VPT.VP.STATE
0128 0014'
012A 2107       LD      R7, #READY
012C 0001
                ! SAVE LOGICAL CPU # !
012E 93FD       PUSH    @R15, R13
0130 5F00       CALL    LIST_INSERT  !R2: OBJ ID
0132 0000*
                                      R3: LIST_PTR ADDR
                                      R4: NEXT_OBJ_PTR
                                      R5: PRIORITY_PTR
                                      R6: STATE_PTR
                                      R7: STATE  !
                ! RESTORE LOGICAL CPU # !
0134 97FD       POP     R13, @R15
               ! PUT CURRENT VP IN READY STATE !
0136 61D2      LD       R2, VPT.RUNNING_LIST(R13)
0138 0002'
013A 4D25      LD       VPT.VP.STATE(R2), #READY
013C 0014'
013E 0001
               ! SET DBR !
0140 612E      LD       R14, VPT.VP.DBR(R2)
0142 0010'

              ! SCHEDULE FIRST ELGIBLE READY VP !
0144 5F00      CALL    GETWORK  !R13:LOGICAL CPU #
0146 0000'
                                 R14:DBR !
              FI

              ! UNLOCK VPT !
0148 4D08     CLR  VPT.LOCK
014A 0000'
              ! UNMASK VECTORED INTERRUPTS !
014C 7C05     EI    VI

014E 9E08     RET
0150       END SIGNAL
```

```
0150        SET PREEMPT        PROCEDURE
           !****************************
            * SETS PREEMPT INTERRUPT ON*
            * TARGET_VP. CALLED BY TC_ *
            * ADVANCE.                 *
            ****************************
            * REGISTER USE:            *
            * PARAMETERS:              *
            *  R1:TARGET_VP_ID (INPUT) *
            * LOCAL VARIABLES          *
            *  R1: VP_INDEX            *
            ****************************!
            ENTRY
             ! NOTE: DESIGNED AS SAFE SEQUENCE SO VPT NEED
               NOT BE LOCKED. !

             ! CONVERT VP_ID TO VP_INDEX !
0150 6112    LD         R2, EXT_VP_LIST(R1)
0152 0210'

             ! TURN ON TGT_VP PREEMPT FLAG !
0154 4D25    LD         VPT.VP.PREEMPT(R2), #ON
0156 0018'
0158 FFFF

             ! ** IF TARGET VP NOT LOCAL
                  ( NOT BOUND TO THIS CPU )
             [IE, IF <<CPU_SEG>>CPU_ID<>VPT.VP.PHYS_CPU(R1)]
             THEN SEND HARDWARE PREEMPT INTERRUPT TO
               VPT.VP.CPU(R1). ** !

015A 9E08    RET
015C        END SET_PREEMPT
```

```
015C           IDLE              PROCEDURE
             !*************************
              * LOADS IDLE DBR ON     *
              * CURRENT VP. CALLED BY *
              * TC_GETWORK.           *
              *************************
              * REGISTER USE          *
              *  GLOBAL VARIABLE      *
              *   R13: LOG CPU #      *
              *   R14: DBR            *
              *  LOCAL VARIABLES:     *
              *   R2: CURRENT_VP      *
              *   R3: TEMP VAR        *
              *   R4: VPT.LOCK ADDR   *
              *   R5: TEMP            *
              *************************!
              ENTRY
               ! GET LOGICAL CPU # !
015C 5F00      CALL       GET_CPU_NO  !RETURNS:


               ! LOAD IDLE DBR ON CURRENT VP !
0174 6103      LD         R3, PRDS.IDLE_VP
0176 0A10'
0178 6135      LD         R5, VPT.VP.DBR(R3)
017A 0010'
017C 6F25      LD         VPT.VP.DBR(R2), R5
017E 0010'

               ! TURN ON CURRENT VP'S IDLE FLAG !
0180 4D25      LD         VPT.VP.IDLE_FLAG(R2), #ON
0182 0016'
0184 FFFF
               ! SET VP TO READY STATE !
0186 4D25      LD         VPT.VP.STATE(R2), #READY
0188 0014'
018A 0001

               ! SCHEDULE FIRST ELIGIBLE READY VP !
018C 5F00      CALL    GETWORK  !R13:LOGICAL CPU #
018E 0000'
                                  R14:DBR !

               ! UNLOCK VPT !
0190 4D08      CLR  VPT.LOCK
0192 0000'
               ! UNMASK VECTORED INTERRUPTS !
0194 7C05      EI     VI

0196 9E08      RET
0198          END IDLE
```

```
0198          SWAP_VDBR        PROCEDURE
             !**************************
              * LOADS NEW DBR ON       *
              * CURRENT VP. CALLED BY  *
              * TC_GETWORK.            *
              **************************
              * REGISTER USE           *
              *  PARAMETERS            *
              *   R1: NEW_DBR (INPUT)  *
              *  GLOBAL VARIABLES      *
              *   R13: LOGICAL CPU #   *
              *   R14: DBR             *
              *  LOCAL VARIABLES       *
              *   R2: CURRENT_VP       *
              *   R4: VPT.LOCK ADDR    *
              **************************!
              ENTRY
               ! SAVE NEW DBR !
0198 93F1      PUSH       @R15, R1
               ! MASK INTERRUPTS !
019A 7C01      DI    VI
               ! LOCK VPT !
019C 7604      LDA        R4, VPT.LOCK
019E 0000'
01A0 5F00      CALL       SPIN_LOCK  ! (R4:^VPT.LOCK) !
01A2 0282'
               ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP.!
               ! GET CPU # !
01A4 5F00      CALL       GET_CPU_NO  !RETURNS:
01A6 02C8'
                                       R1: CPU #
                                       R2:# VP'S!
01A8 A11D      LD         R13, R1
               ! GET CURRENT VP !
01AA 61D2      LD         R2, VPT.RUNNING_LIST(R13)
01AC 0022'
                          ! * * * DEBUG * * * !
01AE 4D21                 CP VPT.VP.MSG_LIST(R2), #NIL
01B0 001E'
01B2 FFFF
01B4 5E06                 IF NE ! MSG WAITING !  THEN
01B6 01C4'
01B8 2100                  LD  R0, #S_N_A  ! SWAP NOT ALLOWED !
01BA 0005
01BC 7601                  LDA R1, $   !PC!
01BE 01BC'
01C0 5F00                  CALL MONITOR
01C2 A900
                          FI
                          ! * * END DEBUG * * !
               ! SET DBR !
```

```
01C4 612E    LD          R14, VPT.VP.DBR(R2)
01C6 0010'
             ! RESTORE NEW DBR !
01C8 97F0    POP         R0, @R15
01CA 5F00    CALL        GET_DBR_ADDR   ! (R0: DBR #)
01CC 000A'
                                             RETURNS
                                            (R1: DBR ADDR) !

             ! LOAD NEW DBR ON CURRENT VP !
01CE 6F21    LD          VPT.VP.DBR(R2), R1
01D0 0010'

             ! TURN OFF IDLE FLAG !
01D2 4D25    LD          VPT.VP.IDLE_FLAG(R2), #OFF
01D4 0016'
01D6 0000

             ! SET VP TO READY STATE !
01D8 4D25    LD          VPT.VP.STATE(R2), #READY
01DA 0014'
01DC 0001

             ! SCHEDULE FIRST ELGIBLE READY VP !
01DE 5F00    CALL   GETWORK  !R13:LOGICAL CPU #
01E0 0000'
                              R14:DBR !

             ! UNLOCK VPT !
01E2 4D08    CLR  VPT.LOCK
01E4 0000'
             ! UNMASK VECTORED INTERRUPTS !
01E6 7C05    EI    VI

01E8 9E08    RET
01EA        END SWAP_VDBR
```

```
01EA            PHYS_PREEMPT_HANDLER   PROCEDURE
               !********************************
                * HARDWARE PREEMPT INTERRUPT   *
                * HANDLER.  ALSO TESTS FOR     *
                * VIRTUAL PREEMPT INTERRUPT    *
                * FLAG AND INVOKES INTERRUPT   *
                * HANDLER IF FLAG IS SET.      *
                * INVOKED UPON EVERY EXIT FROM *
                * KERNEL.  KERNEL FCW MASKS    *
                * NVI INTERRUPTS TO PREVENT    *
                * SIMULTANEOUS PREEMPT INTERR. *
                * HANDLING.                    *
                ********************************
                * REGISTER USE                 *
                *  LOCAL VARIABLES             *
                *   R1: PREEMPT_INT_FLAG       *
                *   R2: CURRENT_VP             *
                * GLOBAL VARIABLES             *
                *   R13:LOGICAL CPU #          *
                *   R14:DBR                    *
                ********************************!

                ENTRY

                 ! * * PREEMPT_HANDLER * * !

                 ! SAVE ALL REGISTERS !
01EA 030F        SUB      R15, #32
01EC 0020
01EE 1CF9        LDM      @R15, R1, #16
01F0 010F

                 ! SAVE NORMAL STACK POINTER (NSP) !
01F2 7D67        LDCTL    R6, NSP
01F4 93F6        PUSH     @R15, R6
                 ! GET CPU # !
01F6 5F00        CALL    GET_CPU_NO !RETURNS:
01F8 02C8'
                                         R1: CPU #
                                         R2:# VP'S!
01FA A11D        LD        R13, R1
                 ! MASK INTERRUPTS !
01FC 7C01        DI    VI
                 ! LOCK VPT !
01FE 7604        LDA    R4, VPT.LOCK
0200 0000'
0202 5F00        CALL  SPIN_LOCK
0204 0282'
                 !RETURNS WHEN VPT IS LOCKED!
                 ! SET DBR !
0206 61D2        LD        R2, VPT.RUNNING_LIST(R13)
```

```
0208 0002'
020A 612E    LD       R14, VPT.VP.DBR(R2)
020C 0010'

             ! PUT CURRENT PROCESS IN READY STATE !
020E 4D25    LD       VPT.VP.STATE(R2), #READY
0210 0014'
0212 0001
0214 5F00    CALL     GETWORK  !R13:LOG CPU #
0216 0000'
                                R14:DBR !
           PREEMPT_RET:
             ! UNLOCK VPT !
0218 4D08    CLR    VPT.LOCK
021A 0000'
             ! UNMASK VECTORED INTERRUPTS !
021C 7C05    EI     VI
           KERNEL_EXIT:
             ! *** UNMASK VIRTUAL PREEMPTS *** !
             ! ** NOTE: SAFE SEQUENCE AND DOES NOT REQUIRE
                      VPT TO BE LOCKED. ** !

             ! GET CURRENT_VP !
021E 610D    LD     R13, PRDS.LOG_CPU_ID
0220 0A0C'
0222 61D2    LD R2, VPT.RUNNING_LIST(R13)
0224 0002'

             ! TEST PREEMPT INTERRUPT FLAG !
0226 4D21    CP       VPT.VP.PREEMPT(R2), #ON
0228 0018'
022A FFFF
022C 5E0E    IF EQ   !  PREEMPT FLAG IS ON !  THEN
022E 0240'

                ! RESET PREEMPT FLAG !
0230 4D25       LD   VPT.VP.PREEMPT(R2), #OFF
0232 0018'
0234 0000
                ! SIMULATE VIRTUAL PREEMPT INTERRUPT !
0236 2101       LD   R1, #0
0238 0020
023A 6112       LD   R2, VPT.VIRT_INT_VEC(R1)
023C 000C'
023E 1E28       JP   @R2
            !NOTE: THIS JUMP TO TRAFFIC_CONTROL
             IS USED ONLY IN THE CASE OF A PREEMPT INTERRUPT,
             AND SIMULATES A HARDWARE INTERRUPT. ** !

             ! *** END VIRTUAL PREEMPT HANDLER *** !
             FI
```

```
                    ! NOTE: SINCE A HDWE INTERRUPT DOES NOT EXIT
                         THROUGH THE GATE, THOSE FUNCTIONS PROVIDED
                           BY A GATE EXIT TO HANDLE PREEMPTS MUST BE
                           PROVIDED HERE ALSO. !

                     ! RESTORE NSP !
0240 97F6            POP R6, @R15
0242 7D6F            LDCTL   NSP, R6
                     ! RESTORE ALL REGSTERS !
0244 1CF1            LDM       R1, @R15, #16
0246 010F
0248 010F            ADD       R15, #32
024A 0020

                     ! EXECUTE HARDWARE INTERRUPT RETURN !
024C 7B00            IRET

024E            END PHYS_PREEMPT_HANDLER
```

```
024E           RUNNING_VP                   PROCEDURE
              !*********************************
               * CALLED BY TRAFFIC CONTROL.     *
               * RETURNS VP_ID. RESULT IS VALID*
               * ONLY WHILE APT IS LOCKED.      *
               **********************************
               * REGISTER USE                   *
               *  PARAMETERS                    *
               *   R1: EXT_VP_ID  (RETURNED)    *
               *   R3: LOG CPU # (RETURNED)     *
               *  LOCAL VARIABLES               *
               *    R2: VP INDEX                *
               *********************************!
               ENTRY
                ! MASK INTERRUPTS !
024E 7C01       DI    VI
                ! LOCK VPT !
0250 7604       LDA        R4, VPT.LOCK
0252 0000'
0254 5F00       CALL       SPIN_LOCK  ! (R4:^VPT.LOCK) !
0256 0282'
                ! NOTE: RETURNS WHEN VPT IS LOCKED BY THIS VP !
                ! GET LOGICAL CPU # !
0258 5F00       CALL       GET_CPU_NO  !RETURNS:
025A 02C8'
                                         R1: CPU #
                                         R2:# VP'S!
025C A113       LD         R3, R1
025E 6132       LD         R2, VPT.RUNNING_LIST(R3)
0260 0002'
                ! CONVERT VP_INDEX TO VP_ID !
0262 6121       LD         R1, VPT.VP.EXT_ID(R2)
0264 0020'
                           ! * * * DEBUG * * * !
0266 0B01                  CP  R1, #NIL
0268 FFFF
026A 5E0E                  IF EQ  ! KERNEL PROC !  THEN
026C 027A'
026E 2100                   LD R0, #V_I_E  ! VP INDEX ERROR !
0270 0006
0272 7601                   LDA R1, $
0274 0272'
0276 5F00                   CALL MONITOR
0278 A900
                           FI
                           ! * * END DEBUG * * !
                ! UNLOCK VPT !
027A 4D08       CLR        VPT.LOCK
027C 0000'
                ! UNMASK VECTORED INTERRUPTS !
027E 7C05       EI    VI
0280 9E08       RET
0282           END RUNNING_VP
```

```
0282               SPIN_LOCK   PROCEDURE
                  !**************************
                   * USES SPIN_LOCK MECH.   *
                   * LOCKS UNLOCKED DATA    *
                   * STRUCTURE (POINTED TO  *
                   * BY INPUT PARAMETER).   *
                   **************************
                   *REGISTER USE            *
                   * PARAMETERS             *
                   *  R4: LOCK ADDR (INPUT)*
                   **************************!
                   ENTRY
                   !  NOTE: SINCE ONLY ONE PROCESSOR CURRENTLY
                            IN SYSTEM, LOCK NOT NECESSARY. ** !
                       ! * * * DEBUG * * * !
0282 0D41          CP   @R4, #OFF
0284 0000
0286 5E06          IF NE ! NOT UNLOCKED !  THEN
0288 0296'
028A 2100           LD  R0, #U_L    ! UNAUTHORIZED LOCK !
028C 0000
028E 7601           LDA R1, $
0290 028E'
0292 5F00           CALL MONITOR
0294 A900
                            FI
                            ! * * END DEBUG * * !
                          TEST_LOCK:
                            !  DO WHILE STRUCTURE LOCKED !
0296 0D46        TSET      @R4
0298 E5FE        JP MI, TEST_LOCK
                            ! ** NOTE SEE PLZ/ASM MANUAL
                                     FOR RESTRICTIONS ON
                                     USE OF TSET. ** !
029A 9E08        RET

029C           END SPIN_LOCK
```

```
029C          ITC_GET_SEG_PTR          PROCEDURE
             !**********************************
              * GETS BASE ADDRESS OF SEGMENT   *
              * INDICATED.                      *
              **********************************
              * REGISTER USE:                   *
              *  R0:SEG BASE ADDRESS(RET)       *
              *  R1:SEG NR (INPUT)              *
              *  R2:RUNNING_VP (LOCAL)          *
              *  R3:DBR_VALUE (LOCAL)           *
              *  R4:VPT.LOCK                    *
              *  R13:LOGICAL CPU #              *
              **********************************!

              ENTRY
               ! SAVE SEGMENT # !
029C 93F1      PUSH    @R15, R1
               ! MASK INTERRUPTS !
029E 7C01      DI      VI
               ! LOCK VPT !
02A0 7604      LDA     R4,VPT.LOCK
02A2 0000'
02A4 5F00      CALL    SPIN_LOCK  !R4:^VPT.LOCK!
02A6 0282'
               ! GET CPU # !
02A8 5F00      CALL    GET_CPU_NO  !RETURNS:
02AA 02C8'
                                    R1: CPU #
                                    R2:# VP'S!
02AC A11D      LD      R13, R1
               ! RESTORE SEGMENT # !
02AE 97F1      POP     R1, @R15
02B0 61D2      LD      R2,VPT.RUNNING_LIST(R13)
02B2 0002'
02B4 6123      LD      R3,VPT.VP.DBR(R2)
02B6 0010'
               ! UNLOCK VPT !
02B8 4D08      CLR     VPT.LOCK
02BA 0000'
               ! UNMASK VECTORED INTERRUPTS !
02BC 7C05      EI      VI
02BE 1900      MULT    RR0,#4
02C0 0004
02C2 7130      LD      R0,R3(R1)
02C4 0100

02C6 9E08      RET
02C8          END ITC_GET_SEG_PTR
```

```
02C8        GET_CPU_NO       PROCEDURE
           !*************************
            * FIND CURRENT CPU_NO    *
            * CALLED BY DIST MMGR    *
            * AND MM                 *
            **************************
            * RETURNS                *
            * R1: CPU_NO             *
            * R2: # OF VP'S          *
            **************************!

            ENTRY
02C8 6101    LD      R1, PRDS.LOG_CPU_ID
02CA 0A0C'
02CC 6102    LD      R2, PRDS.VP_NR
02CE 0A0E'
02D0 9E08    RET
02D2      END GET_CPU_NO


02D2      K_LOCK                PROCEDURE
           !*************************
            *  STUB FOR WAIT LOCK    *
            **************************
            *  R4:^LOCK (INPUT)      *
            **************************!

            ENTRY
02D2 5F00    CALL   SPIN_LOCK
02D4 0282'
02D6 9E08    RET
02D8      END K_LOCK

02D8      K_UNLOCK              PROCEDURE
           !*************************
            *  STUB FOR WAIT UNLOCK  *
            **************************
            *  R4:^LOCK (INPUT)      *
            **************************!

            ENTRY
02D8 0D48    CLR   @R4
02DA 9E08    RET
02DC      END K_UNLOCK

         END INNER_TRAFFIC_CONTROL
```

LIST OF REFERENCES

1. O'Connell, J. S., and Richardson, L. D., Distributed Secure Design for a Multi-Microprocessor Operating System, MS Thesis, Naval Postgraduate School, June 1979.

2. Parks, E. J., The Design of a Secure File Storage System, MS Thesis, Naval Postgraduate School, December 1979.

3. Coleman, A. R., Security Kernel Design for a Microprocessor-Based, Multilevel, Archival Storage System, MS Thesis, Naval Postgraduate School, December 1979.

4. Moore, E. E. and Gary, A. V., The Design and Implementation of the Memory Manager for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, June 1980.

5. Reitz, S. L., An Implementation of Multiprograming and Process Management for a Security Kernel Operating System, MS Thesis, Naval Postgraduate School, June 1980.

6. Wells, J. T., Implementation of Segment Management for a Secure Archival Storage System, MS Thesis, Naval Postgraduate School, September 1980.

7. Organick, E. J., The Multics System: An Examination of Its Structure, MIT Press, 1972.

8. Madnick, S. E., and Donovan, J. J., Operating Systems, McGraw Hill, 1974.

9. Reed, P. D., Processor Multiplexing In a Layered Operating System, MS Thesis, Massachusetts Institute of Technology, MIT LCS/TR-167, 1979.

10. Schell, Lt.Col. R. R., "Computer Security: the Achilles Heel of the Electronic Air Force?," Air University Review, v. 30, no. 2, p. 16-33, January 1979.

11. Schell, Lt.Col. R. R., "Security Kernels: A Methodical Design of System Security," USE Technical Papers (Spring Conference, 1979), p. 245-250, March 1979.

12. Denning, D. E., "A Lattice Model of Secure Information Flow," Communications of the ACM, v. 19, p. 236-242, May 1976.

13. Schroeder, M. D., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM, v. 15, no. 3, p. 157-170, March 1972.

14. Dijkstra, E. W., "The Humble Programmer," Communications of the ACM, v. 15, no. 10, p. 859-865, October 1972.

15. Reed, D. P., and Kanodia, R. K., "Synchronization With Eventcounts and Sequencers," Communications of the ACM, v. 22, no. 2, p. 115-124, February 1979.

16. Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Ph.D. Thesis, Massachusetts Institute of Technology, 1966.

17. Zilog, Inc., Z8001 CPU Z8002 CPU, Preliminary Product Specification, March 1979.

18. Zilog, Inc., Z8010 MMU Memory Management Unit, Preliminary Product Specification, October 1979.

19. Advanced Micro Computers, AM96/4116 AmZ8000 16-Bit MonoBoard Computer, User's Manual, 1980.

20. Zilog, Inc., Z8000 PLZ/ASM Assembly Language Programming Manual, 03-3055-01, Revision A, April 1979.

21. Schell, R. R. and Cox, L. A., Secure Archival Storage System, Part I - Design, Naval Postgraduate School, NPS52-80-002, March 1980.

22. Schell, R. R. and Cox, L. A., Secure Archival Storage System, Part II - Segment and Process Management Implementation, Naval Postgraduate School, NPS52-81-001, March 1981.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center ATTN:DDC-TC Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
4.	LTCOL Roger R. Schell, Code 52Sj Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
5.	Lyle A. Cox, Jr., Code 52Cl Department of Computer Science Naval Postgraduate School Monterey, California 93940	4
6.	Joel Trimble, Code 221 Office of Naval Research 800 North Quincy Arlington, Virginia 22217	1
7.	Department Chairman Department of Computer Science United States Military Academy West Point, New York 10996	1
8.	INTEL Corporation Attn: Mr. Robert Childs Mail Code: SC 4-490 3065 Bowers Avenue Santa Clara, California 95051	1
9.	John P.L. Woodward The MITRE Corporation P.O. Box 208 Bedford, Massachusetts 01730	1

10.	Digital Equipment Corporation Attn: Mr. Donald Gaubatz 146 Main Street MI 3-2/E41 Maynard, Massachusetts 01754	1
11.	Joe Urban University of Southwestern Louisiana P.O. Box 44330 Lafayette, Louisiana 70504	1
12.	LCDR Gary Baker, Code 37 Department of Computer Technology Naval Postgraduate School Monterey, California 93943	1
13.	LCDR John T. Wells P.O. Box 366 Waynesboro, Mississippi 39367	1
14.	CPT Anthony R. Strickler Route #12 West Sipley Ferry Road Kingsport, Tennessee 37663	5